INTERNAL MAINTENANCE SPECIFICATION

FORTRAN V5

77987506A

0148P-0030A/0103P

Section A: Overview

Section B: Deck and Routine Descriptions

A. OVERVIEW

This section is provided to give general information about FTN5. It is organized as follows:

1. Compiler Structure.

2. Global Data Structures.

3. COMDECKS.

1.0 COMPILER STRUCTURE

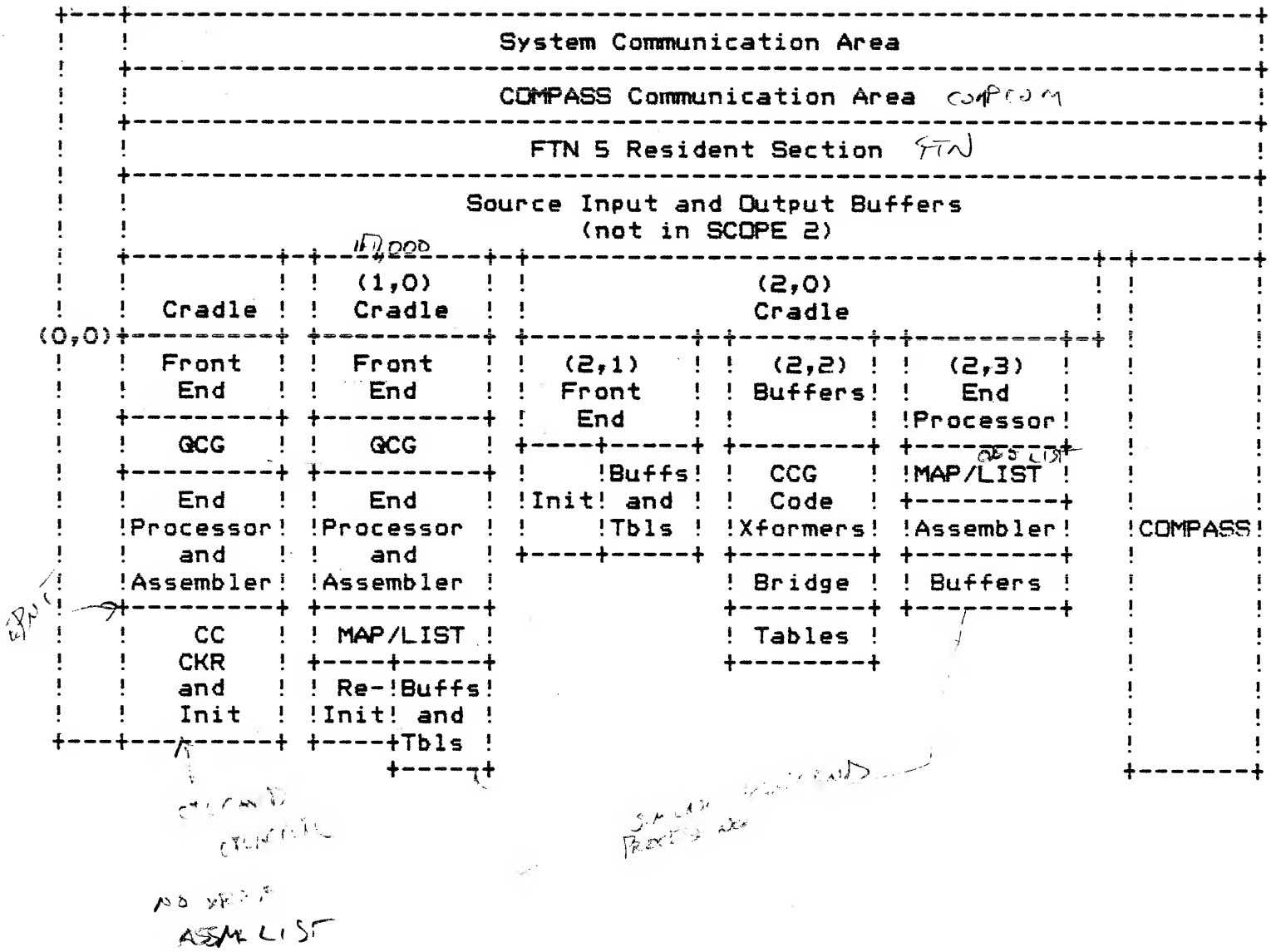
The FORTRAN 5 (FTN5) compiler consists of four optimization levels (OPT=0,1,2,3) divided into quick mode (QCG) (OPT=0) and optimized mode (CCG) (OPT>0). QCG and CCG utilize a common front end (lexical, syntactic and semantic analysis), but have distinct code generation mechanisms. A common rear end (assembler, binary output, object listing, map and cross reference listing) is used by all levels of optimization.

The common front/rear end processing with distinct code generators is implemented as an overlay structure. The FTN5 overlays and their principal contents are shown in Figure 1.

Note that the entire QCG mode compiler is included in the (0,0) overlay. The (1,0) overlay (also containing the entire QCG compiler) is loaded when OPT=0 is requested but object listing and/or map listing function are required or when intermixed COMPASS assembly is required. If OPT>0 is requested, the (2,0) overlay is loaded, followed by successive loads of the (2,1), (2,2) and (2,3) overlays for each program unit encountered. Intermixed COMPASS when OPT>0 requires a reload of the (2,0) overlay.

The compiler can be split into logical functions, reflected in the overlay structure. The functions are cradle, front end, QCG, CCG, rear end and frame. These are described below.

FTN5 source code resides on an UPDATE PL.


System Communication Area
COMPASS Communication Area
FTN 5 Resident Section
Source Input and Output Buffers
(not in SCOPE 2)
(0,0)
Cradle
Front End
QCG
End Processor and Assembler
CC CKR and Init
(1,0) Cradle
Front End
QCG
End Processor and Assembler
MAP/LIST
Re-Init
Buffs and Tbls
(2,0) Cradle
(2,1) Front End
Init
Buffs and Tbls
(2,2) Buffers
CCG Code Xformers
Bridge Tables
(2,3) End Processor
MAP/LIST Assembler
Buffers
COMPASS

1.1 Cradle: The cradle consists of support routines which are used by all overlays. Included here are routines/functions which are present in some form for all or most overlays. The following decks are included:

FTN: Tables and cells which survive overlay loads. Overlay processing routines.

UTILITY: General utility routines (mostly comdecks) for conversion, I/O, etc.

PUC: Program unit controller. Table control vectors, global cells, control routines, file closing, statistics.

LINK: Routines to support common front/rear ends. Provides stub routines for functions not required by one of the code generation modes.

QCGLINK	:	(0,0) and (1,0) overlays
CCGLINK	:	(2,0) overlay
ZEROLINK	:	(0,0) overlay
RLINK	:	(2,3) overlay
FLINK	:	(2,1) overlay

PEM: Print error messages. Routines to output diagnostic texts (not the texts). Not present in the (2,2) overlay.

ALLOC: Table management and allocation. Not present in the (2,2) overlay.

SNAP: Test mode only. The snap formatting and IDP interfaces.

FSNAP	:	(0,0), (1,0) and (2,1) overlay
CSNAP	:	(2,2) overlay
RSNAP	:	(2,3) overlay

IDP: Test mode only. Interactive debug package.

Init: Each overlay (primary and secondary) contains an overlay initialization routine which performs the initializations required by that overlay.

INIT00	:	(0,0)	overlay
INIT10	:	(1,0)	overlay
INIT20	:	(2,0)	overlay
INIT21	:	(2,1)	overlay
INIT22	:	(2,2)	overlay
INIT23	:	(2,3)	overlay

All initialization routines are used once per overlay load and the space they occupy is free for table usage (or whatever).

1.2 Front End: The front end routines provide lexical, syntactic and semantic analysis for the FTN5 compiler. The following decks are included:

FEC: Front end controller. Main control routines for the front end, front end global cells and tables, table search and entry routines.

FERRS: Front end diagnostic texts.

LEX: Lexical analysis (scanner).

HEADER: Header statement processing.

KEY: Keyword statement processing.

CDDIR: C$ directive processing.

DATA: DATA statement processing.

DECL: Declarative statement processing.

TYPE: Explicit and implicit type declaration processing.

FMT: Format statement processing.

IO: I/O statement processing.

PAR: Syntax and semantic processing of expressions (parser).

CONRED: Constant reduction. Compile time arithmetic.

STMTF: Statement function processing.

LABEL: Statement label and DO statement processing.

FSKEL: Front end code skeleton linkage to CSKEL. (2,1) overlay only.

1.3 QCG: The QCG (Quick Code Generator) routines provide the OPT=0 code generation functions. Decks are:

QCGC: QCG controller. Control routines, cells and tables.

QSKEL: QCG instruction skeletons.

FUN: External procedure call and argument processing.

REG: Register selection.

GEN: Main QCG code generation processing.

1.4 CCG: The CCG (common code generator) routines provide the OPT>0 code generation function. Decks are:

CCGC: CCG controller. Control routines, cells and tables.

BRIDGE: Transforms front end IL output to a form understood by the common code generator.

INTERMEDIATE CODE → RLIST

CSKEL: Bridge instruction skeletons.

CCG: The actual common code generator. Decks making up this portion are:

CGTM: Code generation table manager.
MIO: Mass storage I/O routines.
FBV: Form bit vectors.
GPO: Global program optimization.
GRA: Global register assignment.
SQZ: Redundant operation elimination.
MCG: Machine code generation.
BDT: Form dependency tree (graph).
CFA: Control flow analysis.
UDT: Usage/definition table processing.
PROSEQ: Process accumulated sequences.
Output: CCG output routines.

1.5 Rear End: The rear end routines provide assembly, binary output and end time listing functions. The decks are:

REC: Rear end controller. Control routines, tables and cells for the rear end processor.

RERRS: Rear end diagnostic texts.

FAS: FORTRAN assembler. Binary production.

MAP: Attribute, storage map and cross reference listing production.

LIST: Object listing production.

1.6 Frame: A loader interface module. Consists of deck stubs which are place holders for the various decks making up each overlay. The mechanism by which the front and rear ends can be truly common.

2.0 GLOBAL DATA STRUCTURES

B13

This section describes the data structures used by FTN5. Reference to tables will be by the FWA cell (T.xxx). The structures described are classified as follows:

2.1 Global (all overlays and compilation modes)

2.2 Front end

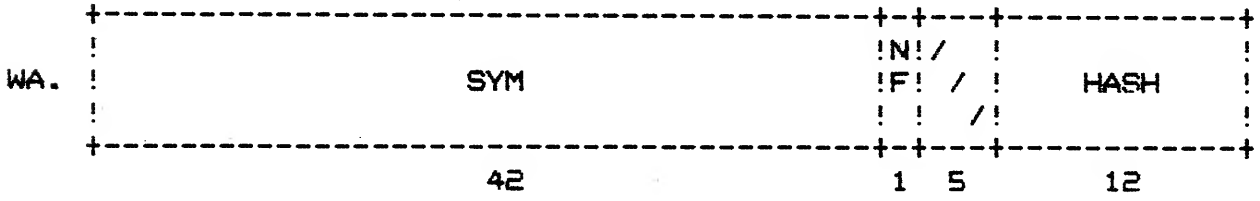
2.3 Special structures

B14

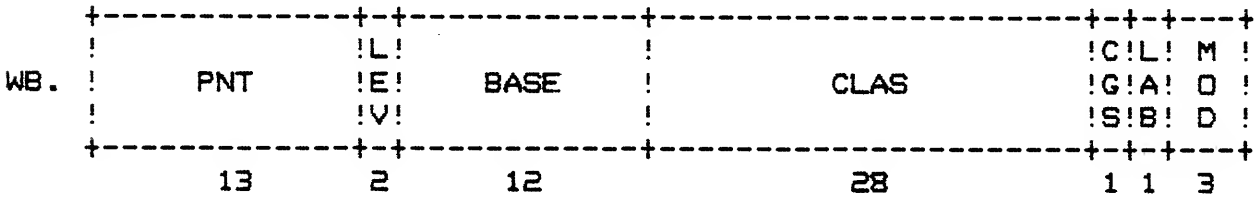
2.1.1 Symbol Table (T.XYM)

The FTN5 symbol table contains source symbols and labels and compiler generated symbols. There are two basic formats, symbols and labels and many subformats, depending on the nature of the symbol/label. The symbol table is dense and entries occur in the order they are encountered/generated during compilation. Access is via hashing and associated with the symbol table is table of hash buckets. A symbol table entry consists of three words, designated WA., WB., WC.

Standard Symbol Format (nonlabel)



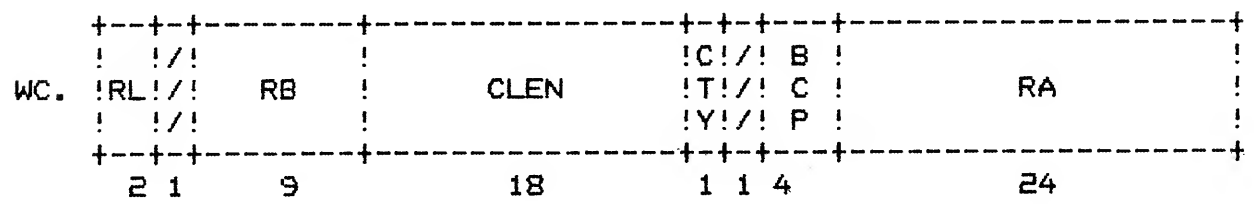
SYM: DPC (left justified, zero filled)
NF: Cannot be formal parameter
HASH: Hash chain pointer (zero if end of chain)



PNT: Pointer field (to T.DIM if array, T.NLST if namelist)
LEV: Level number
BASE: Symbol table ordinal of equivalence class base members
CLAS: Attribute bits (see below)
CGS: Compiler generated symbol
LAB: Statement label
MODE: Mode (type) of symbol

WB.CLAS Bits

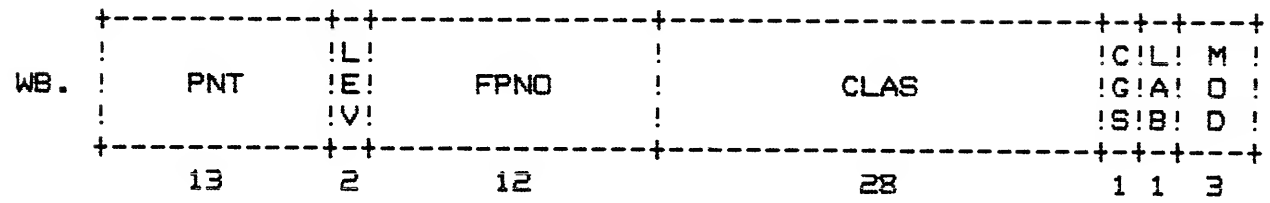
SFA:	Statement function dummy argument
1REF:	One reference (stray name)
MAT:	Materialized (storage required)
SAVE:	Declared in SAVE statement (implicit or explicit)
NLST:	Namelist group name
LEV:	Declared in LEVEL statement (implicit or explicit)
VDS:	Appeared in variable dimension description
TYP:	Appeared in an explicit type declaration
AGN:	Variable in ASSIGN statement
INTF:	Declared in INTRINSIC statement
DEXT:	Declared in EXTERNAL statement
GENF:	Generic function name
LDO:	Load only variable. Also defined as: SFX: used when parsing statement function AGO2: object of assigned goto
BMEM:	Base member at an equivalence class
LOCF:	LOCF function (irreducible)
LCM:	Resides in large core
FP:	Formal parameter (dummy argument)
COM:	In a common block
EXT:	External name
ENT:	Entry point name
FUN:	Function name
SUB:	Subroutine name
ARY:	Array name
EQV:	In an equivalence class
PARM:	Parameter (symbolic constant)
DEF:	Defined (VAR-stored into; SUB/FUN-arg count determined)
NVAR:	Not a variable
VAR:	Variable



RL:	Relocation type
RB:	Relocation block number
CLEN:	Character length (per element)
CTYP:	Character type (constant or assumed length)
BCP:	Beginning character position
RA:	Block relative address

Special Symbol Formats: (WA. is standard)

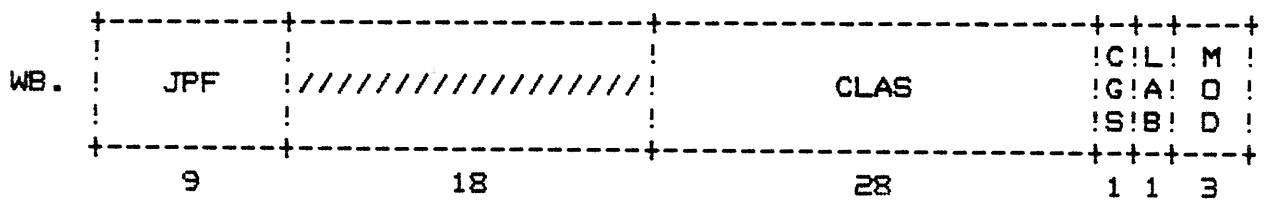
Formal Parameter (Dummy Argument)



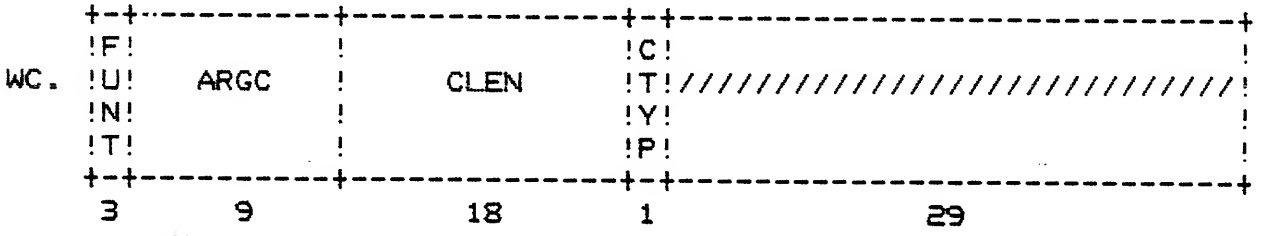
PNT: Standard
LEV: Standard
FPNO: Formal Parameter Number
CLAS: Standard (FP bit on)
CGS: Standard
LAB: Standard
MODE: Standard

WC. is standard

Function/Subroutine (except statement functions)

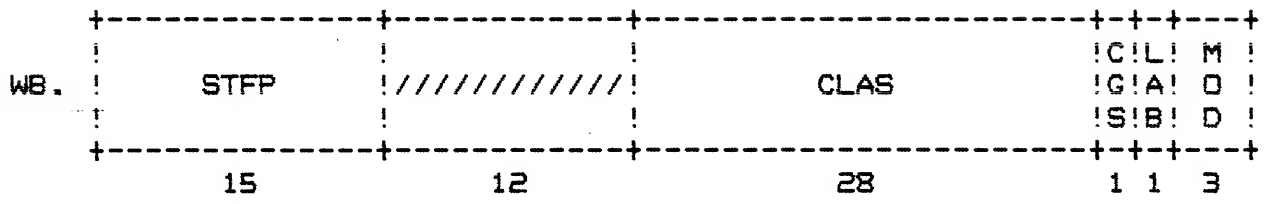


JPF: Index into intrinsic function table (if intrinsic)
CLAS: Standard (with proper bits set)
CGS: Standard
LAB: Standard
MODE: Standard (only on functions)



FUNT: Function type
ARGC: Argument count
CLEN: Standard (function only)
CTYP: Standard (function only)

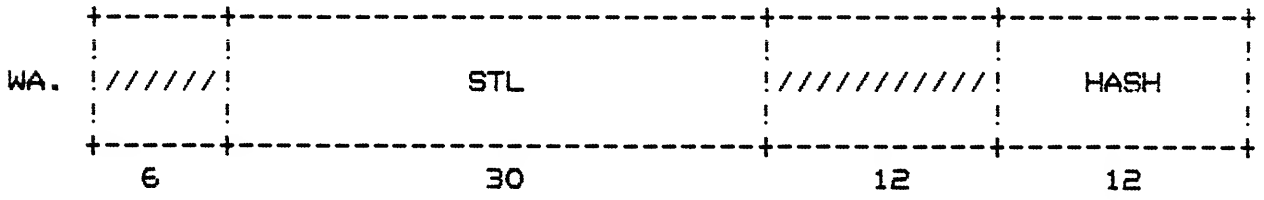
Statement Function



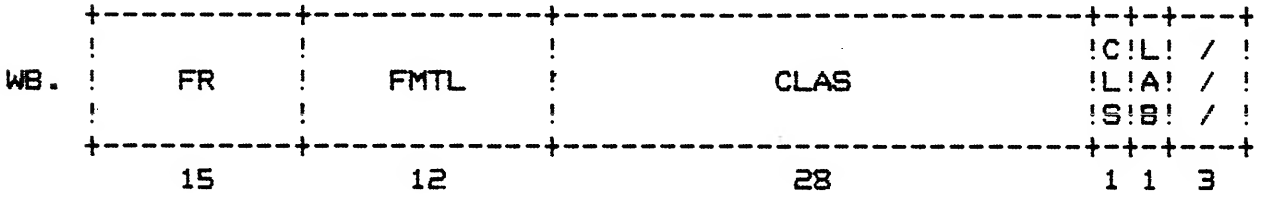
STFP: Pointer to statement function definition (on T.STF)
CLAS: Standard
CGS: Standard
LAB: Standard
MODE: Standard

WC. is in function/subroutine format

Statement Label Format



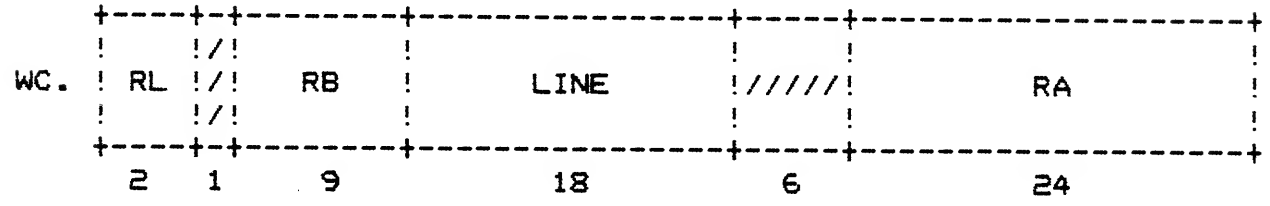
STL: DPC label (right justified, zero filled)
HASH: Standard



FR: Label first reference line number
FMTL: Format length (FDEF), in characters
CLAS: Attribute bits (see below)
CGS: Standard
LAB: Standard (on)

WB.CLAS bits

1REF:	Stray label
INDO:	Label in do loop, loop has exit
NIN:	Do loop has (possible) negative increment
DLPE:	Label is possible do loop entry
DLC:	Do loop has been closed
LC:	CCG internal label
NDEF:	Label defined on non-executable statement
UDEF:	Label undefined
FREF:	Label referenced as format
FDEF:	Label defined on format statement
PRD:	Do parameter redefined in this loop
DLBB:	Loop contains backward branch
DLEN:	Loop contains an entry
DLEX:	Loop contains an exit
DLNI:	Loop not innermost (of a nesting)
DLER:	Loop contains external references
DOGL:	Generated label for do top
DMAT:	Loop index to materialize
ACT:	Label is active
INA:	Label is inactive (cannot be referenced)
SLEN:	Entry to a do loop
SLEX:	Exit from a do loop
SDEF:	Label defined on executable statement
SREF:	Label referenced as executable statement
DOT:	Label is a do loop terminal

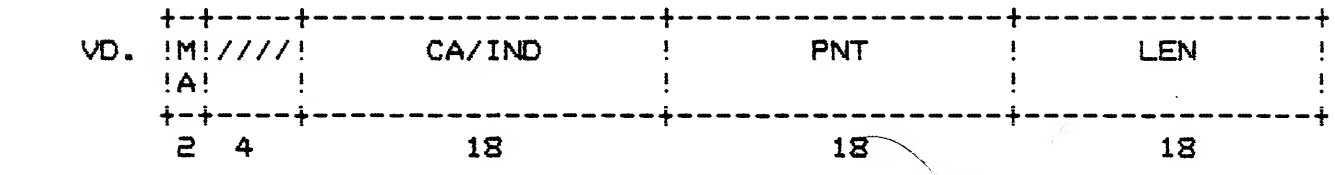


RL:	Standard
RB:	Standard
LINE:	Line number of definition
RA:	Standard

2.1.2 Variable dimension Information Table (T.VDI)

C3

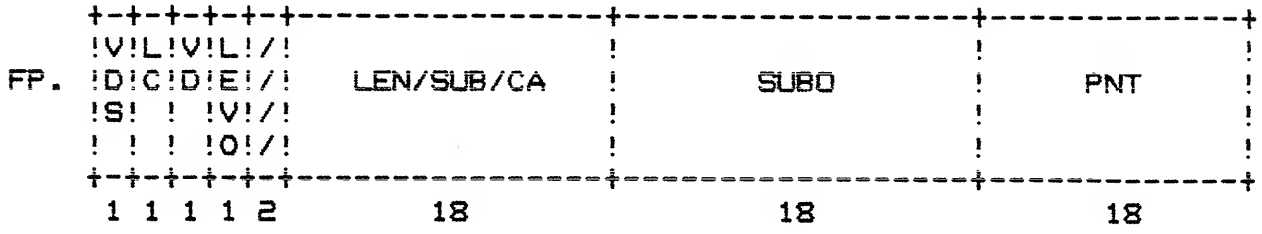
One word table entry for each variable (adjustable) dimension calculation required.



MA: Materialize/Allow flags
CA: Bias for cells (CCG)
IND: Index of VD. store operand (FE)
PNT: Ordinal to vardim turple table (first turple)
LEN: Number of turples

2.1.3 Formal Parameter Information Table (T.FPI)

One word table for each unique formal parameter (dummy argument) in the program unit (the master copy for entry point parameter lists).



VDS: Formal parameter used in variable dimension descr.
LC: CCG made local copy
VD: Used in issued variable dimension
LEVO: If LEVEL 0
LEN: Number of subreferences (FE)
SUB: Index into subtable (end of program unit)
CA: Bias of local copy (CCG)
SUBO: Number of LEVEL 0 references
PNT: Symbol table ordinal

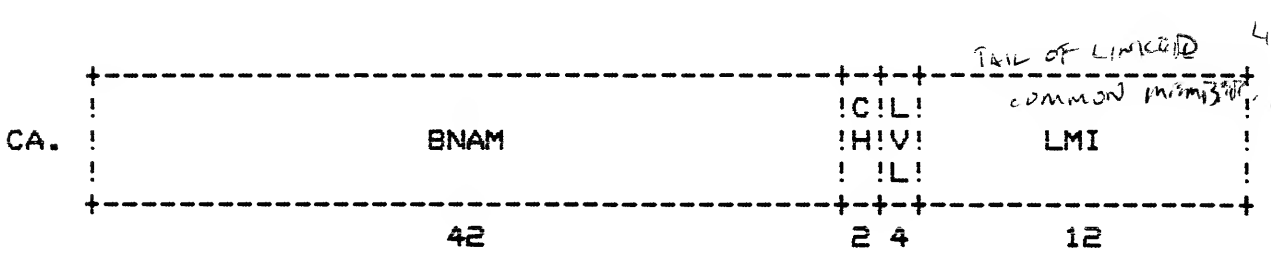
2.1.4 Constant Table (T.CON)

The constant table is unformatted. It contains the binary values of converted (or manufactured) constants encountered during front end processing.

2.1.5 Common Block Name Table (T.BLKS)

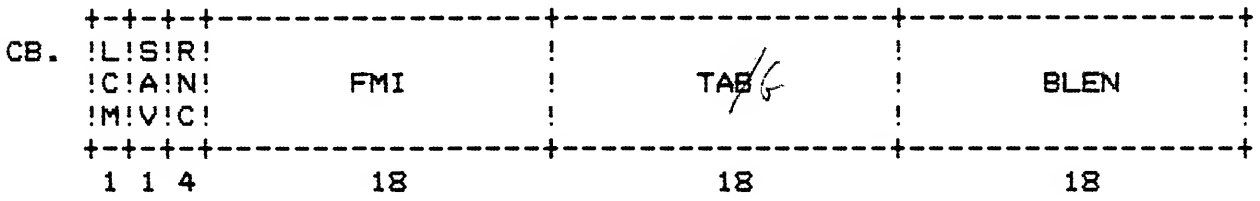
The common block name table consists of one (two word) entry for each named common block (or blank common) encountered during declarative processing.

CA.



BNAM: Block name (DPC, left justified, zero filled)
CH: Character information
CHAR: Block has character entries
NAC: Block has character conflict
LVL: Level information
CNFL: Level conflict
BFLT: Level default
BLFL: Level number
LMI: Index of last member of block in T.COMM

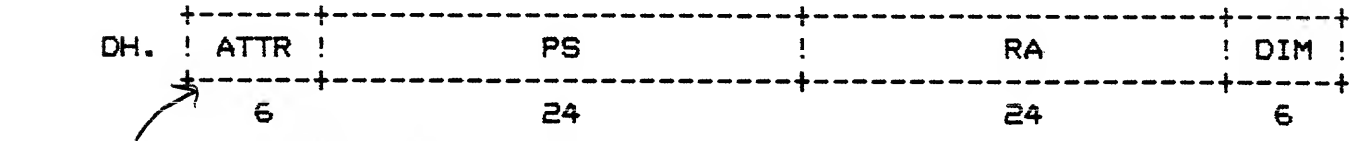
CB.



LCM: Block is in ECS/LCM
SAV: Block appeared in SAVE statement
RNC: Remaining character count before block round
FMI: Index of first member of block in T.COMM
TAG: ECS/LCM pointer tag
BLEN: Block length (words, rounded)

2.1.6 Dimension Table (T.DIM)

The dimension table consists of a header word, and pairs of dimension information, one for each dimension. Only unique dimension information is kept, so if two arrays are dimensioned the same (number and form of each dimension the same) only one entry in T.DIM will be made.

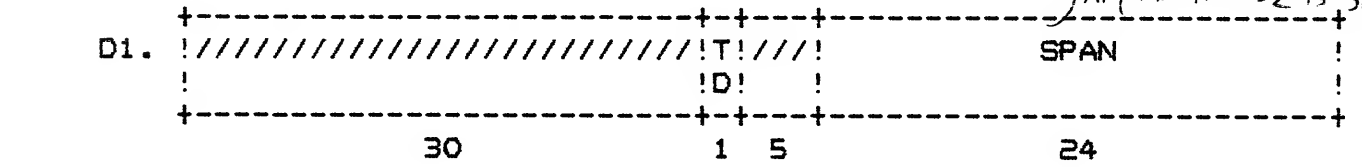


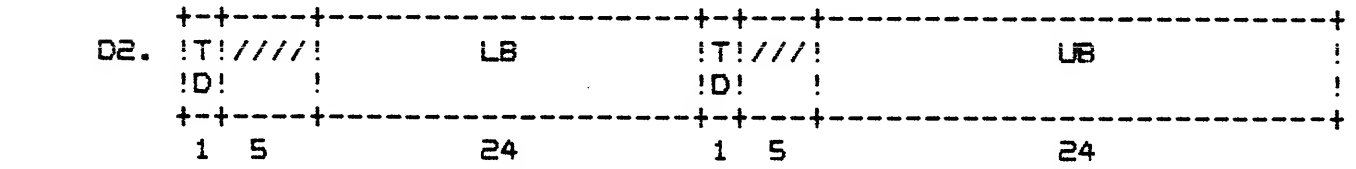
PNTD TO BY SYMTAB PNTFLD

ATTR: Attributes (see below)
PS: Product of spans (partial product for VD) TOTAL # OR ARRAY ELMS
RA: Relative address of runtime dimension table → 5600 TABLE
DIM: Number of dimensions

DH.ATTR bits
- VD: Variable dimension present
- AS: Assumed size array
- VP: Variable product of spans
- MAT: Runtime dimension table materialize

INDEX (UP.+) UNRDIMTBL IS ORDINAL APPTD IN D2 IS SET





TD: Type dimension (on-variable, off-constant)
SPAN: UB-LB+1
UB: Upper bound
LB: Lower bound

2.1.7 Intermediate Language (T.PAR)

The Intermediate Language (IL) is the internal parsed representation of a FTN5 source program. The IL consists of triples (historically misspelled turples), each turple consisting of a header and two operand words. Descriptions of the header and operand words follow:

TH. - Turple header. This general format goes through several transformations, depending upon where the operator is in the parse process:

Operator Selected During Parser Analysis

TH.	SKEL	1ATR	!Q! /// ! !A! /// ! !T! /// ! !R! /// !	STPR	TBPR
	18	14	4 6	9	9

Operator Selected During Parser Synthesis

TH.	SKEL	1ATR	!Q!/! M ! !A!/! O ! !T!/! D ! !R!/! E !	MODC	TBPR
	18	14	4 2 4	9	9

Turple output to T.PAR (IL)

TH.	SKEL	LINE	!Q!C! M ! !A!A! O ! !T!T! D ! !R!R! E !	/////////	TBPR
	18	14	4 2 4	9	9

SKEL: Code skeleton index or relative address
1ATR: Parse attributes (see below)
LINE: Source line number (if first turple of line)
QATR: QCG attributes (see below)
CATR: CCG attributes (see below)
MODE: Type of result (dominant mode)
MODC: Mode coersion
STPR: Stack priority
TBPR: Token buffer priority index

TH.1ATR attribute bits:

NSQZ:	Turple is not squeezable
UNAR:	Unary operator
MDLS:	No associated type
DIS:	Operator is algebraically distributive
COM:	Operator is algebraically commutative
AS:	Operator is algebraically associative
MASK:	Masking/logical operator
CHAR:	Character operands allowed
SMD:	Specific mode determined
BND:	Operator OK for dimension bound expression

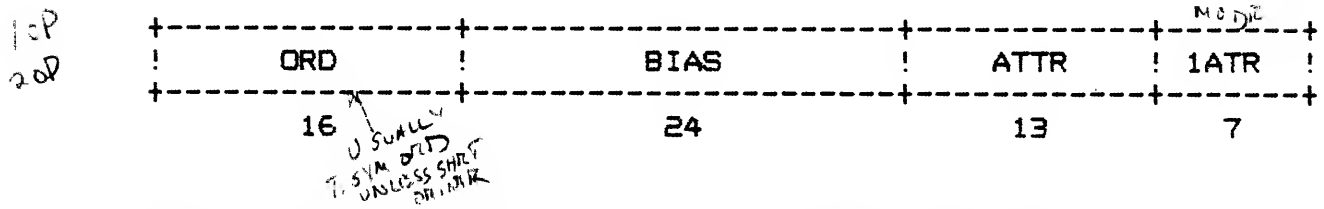
TH.QATR attribute bits:

NSTD:	SKEL field contains routine address
PLC:	Operand is // of passed length item
1DUC:	First operand is register allocated
2DUC:	Second operand is register allocated

TH.CATR attribute bits:

PAP:	First turple of an argument
PFP:	First PAP of an argument list

Operand Format (TP.)



ORD:	Symbol table ordinal of operand (usually)
BIAS:	Constant addend
ATTR:	Attributes (see below)
1ATR:	Parser attributes (see below)

TP.ATTR attribute bits: These bits determine how the ORD and BIAS fields are to be interpreted.

LCM:	Operand is ECS/LCM resident
FP:	Operand is a formal parameter
EQV:	Operand is equivalenced
GL:	Generated label (ORD is GL number)
ARR:	Operand is an array
SHRT:	Short constant (ORD is null, BIAS=constant)
ADDR:	Address reference operand
INTR:	Intermediate operand (ORD is IL pointer)
CAT:	Concatenation operand
IOD:	I/O definition
IOP:	Potential I/O definition

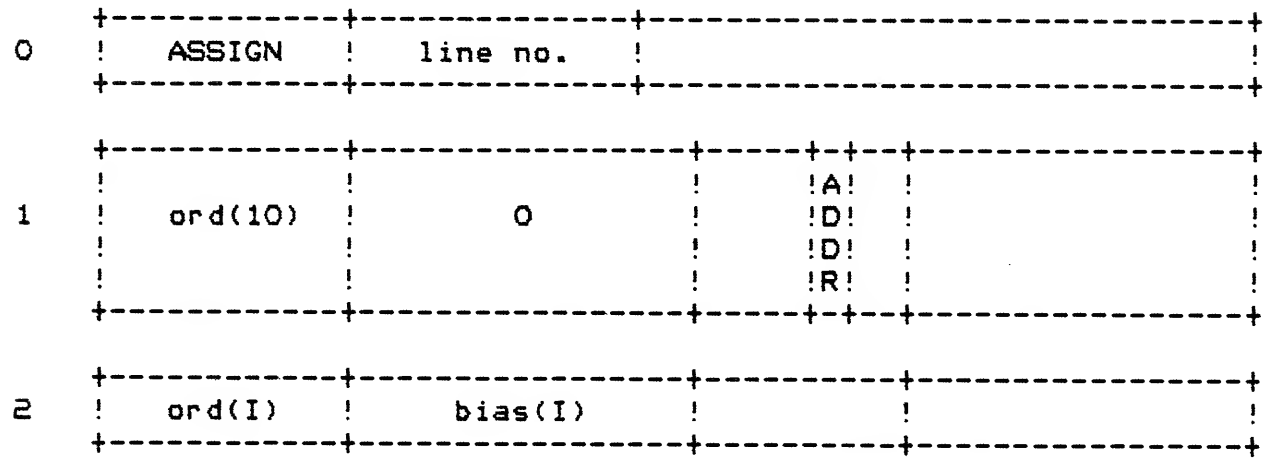
TP.1ATR attribute bits: These bits are used during parse only and are discarded when IL is output.

ARS: Array subscript operand
ARE: Entire array reference
LCF: Reference for LOCF intrinsic
EXPR: Operand was an expression
MODE: Type of operand (3 bits)

Turple Formats

Assign turple

ASSIGN 10 to I will produce



DO-Begin_Turple

The DO statement has the form

DO s i = m1, m2, m3

This statement generates a turple of three triples which contain the required parameters for code generation of a DO-BEGIN. There are two sets of DO-Begin turples: DOBEGAO, DOBEGBO, DOBEGCO for zero trip DO loops, and DOBEGA1, DOBEGB1, DOBEGC1 for one trip DO loops.

0	DO-BEGA OPCODE	Line No.	Mode of i	
1	Operand i			
2	Operand m1			
3	DO-BEGB OPCODE	Line No.	0	
4	Operand m2			
5	Operand m3			
6	DO-BEGC OPCODE	Line No.	0	
7	DO-START label			
8	DO-END label			

Each operand m1, m2, m3, and i appears in the standard Ordinal-Bias format.

The DO-START_label is a generated label ordinal for the zero trip loop or zero if the loop will go at least one trip.

The compiler provides both a control card option and a compile-time directive to compile DO loops as one trip loops, the status of which determines DOBEG op-codes.

When the label associated with the end of the DO loop is encountered the following turple of two triples will be generated.

DO-END-Turple

There are two sets of DO-END turples, DOENDA0, DOENDB0 for zero trip Do loops and DOENDA1, DOENDB1 for one trip Do loops. The trip count is established at DO-BEGIN time.

0	DO-ENDA OPCODE	Line No.	Mode of i	
1	Operand i			
2	Operand m1 (if m3 is constant or simple variable, else zero)			
3	DO-ENDB OPCODE	Line No.		
4	DO-START label			
5	DO-END label			

The presence of Operand m3 is not necessary but may make it more convenient for an optimizer to eliminate the copy code generated in the prologue (Temp=m3).

Arithmetic-IF-Turple

The arithmetic-IF statement has the form

```
IF(e) n1,n2,n3
```

A turple of two triples is required to represent it because there are four input parameters.

	ARITH-IF1 OPCODE	Line No.	Mode of e	
	Operand e			
	Label n1			
	ARITH-IF2 OPCODE	Line No.	0	
	Label n2			
	Label n3			

Logical_IE_Turples

These turples represent

IF(e)s

where e is a logical expression and
s is an executable statement.

A turple of two triples is used to express this construct, with operand-1 of the first turple pointing to the expression and operand-w of the second turple containing the generated label of the branch-around-s target. The op-code is IF.NOTL, since the turple denotes IF (.NOT.e) GO TO gl

The special case

If(e) GO TO lab

will be recognized and produce an L.IF turple with operand-2 of the second triple containing lab.

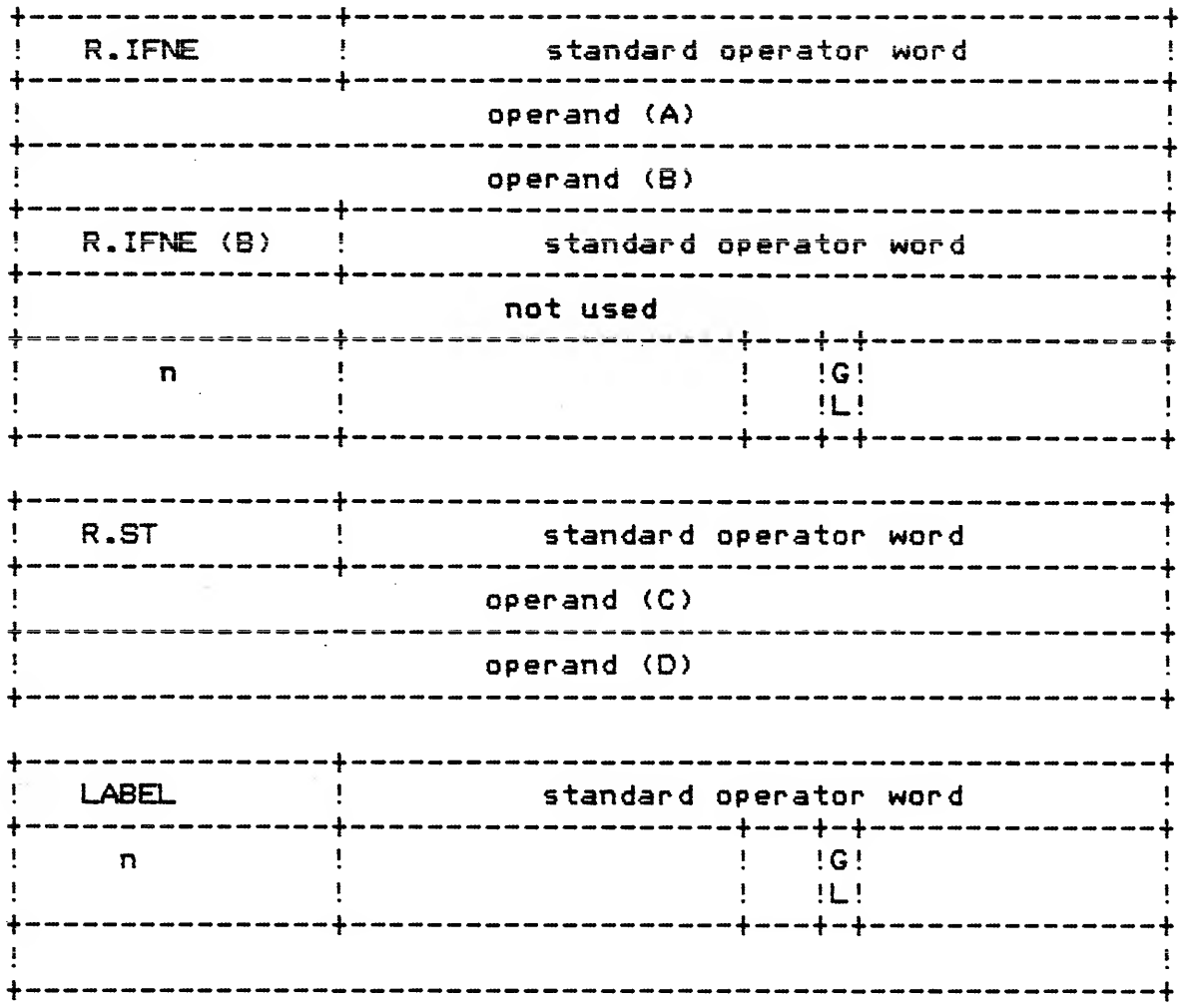
The special case

IF(e) RETURN

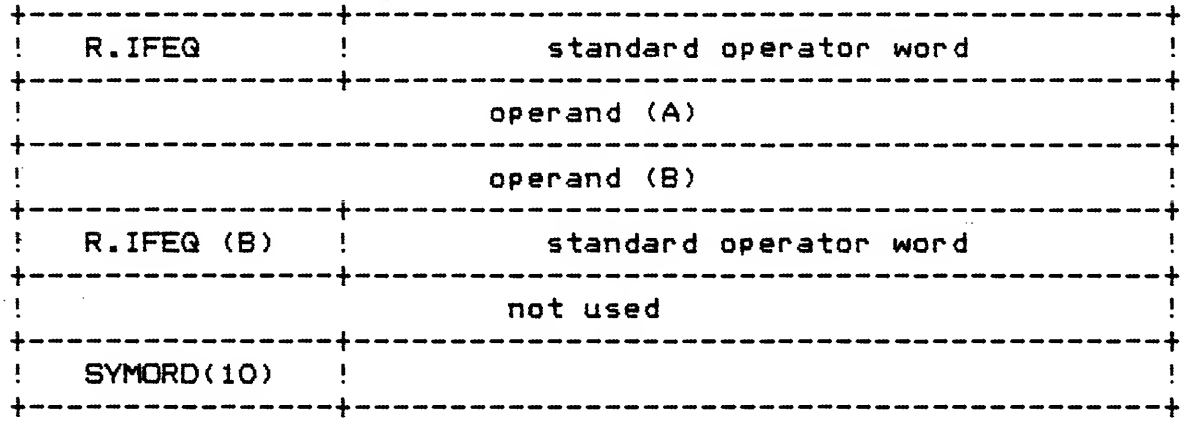
is similar, with the label being at the exit line.

Special turples are used for logical IF statements of one relational operation. The m.IFop turples are 6 words, with the operands of the relational as operands-1 and -2 (words 2 and 3). Word 6 contains the branch target.

IF(A.EQ.B) C=D will produce:

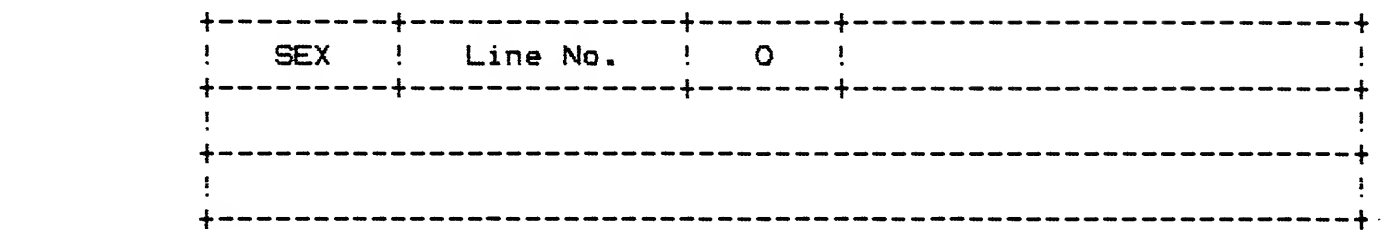


IF(A.EQ.B) GO TO 10 will produce:



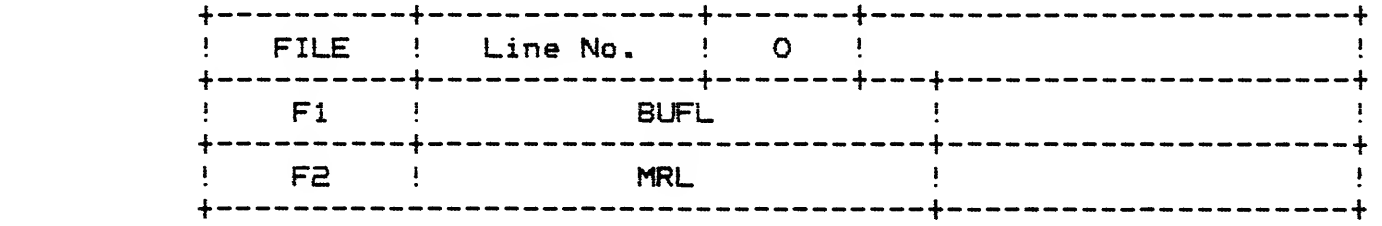
Start Execution (SEX) Turple

This is issued only for a main program.



File Turple

These are issued due to files on a PROGRAM statement and always precede the SEX turple.



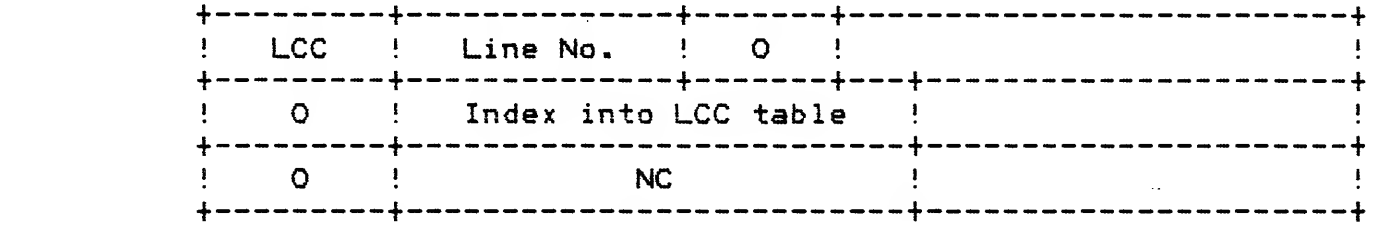
Where: F1 = ordinal into file name table

F2 = ordinal into file name table of equivalenced file (F1 = F2)

BUFL = buffer length value

MRL = maximum record length value

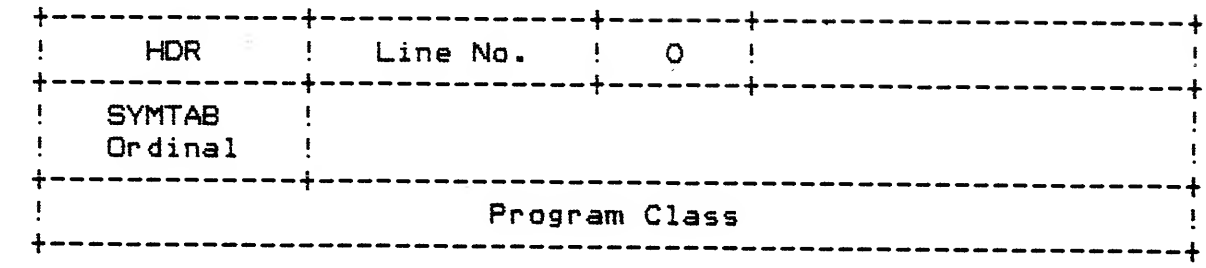
Loader Control Card (LCC) Turple



The LCC table contains the loader directive image to be issued by the assembly phase. NC = number of characters in the directive image. This turple always precedes the SEX turple.

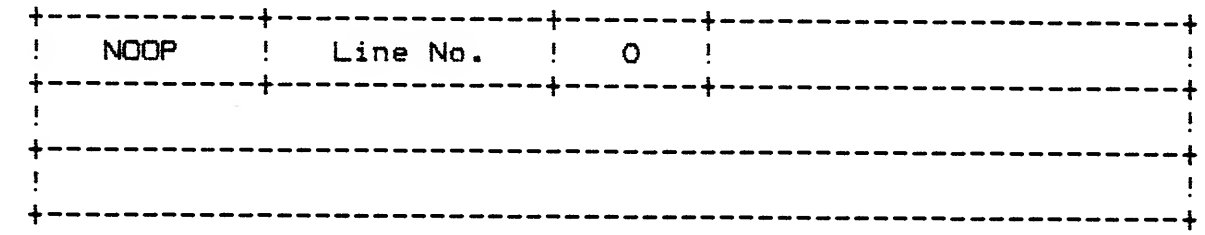
Header (HDR) Turple

This is the first turple in the IL.



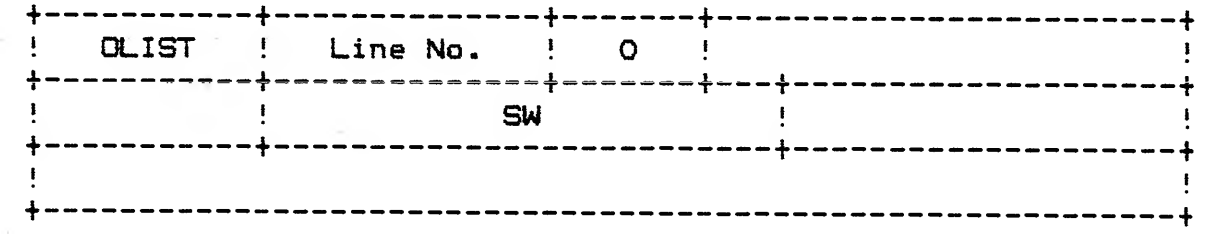
Where: SYMTAB ordinal is the symbol table entry for the routine name. Program class specifies the kind of routine.

NOOP Turple



This turple signifies no actions. It may be a convenient place to put a line number.

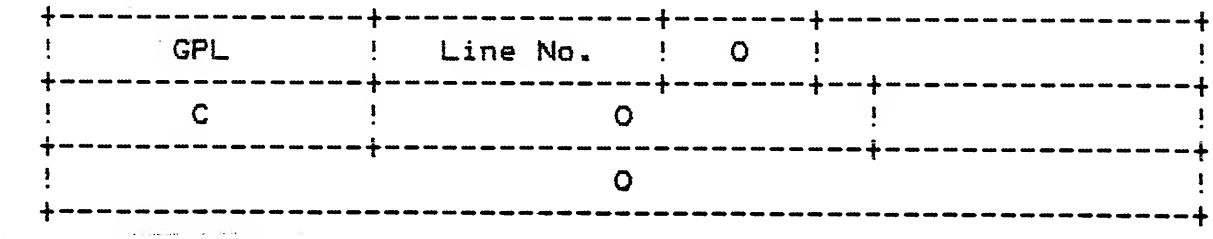
Object Listing Control Turple



Where: SW=1 if object listing is to be turned on and SW=0 if object listing is to be turned off. It is issued only in response to a C$ directive.

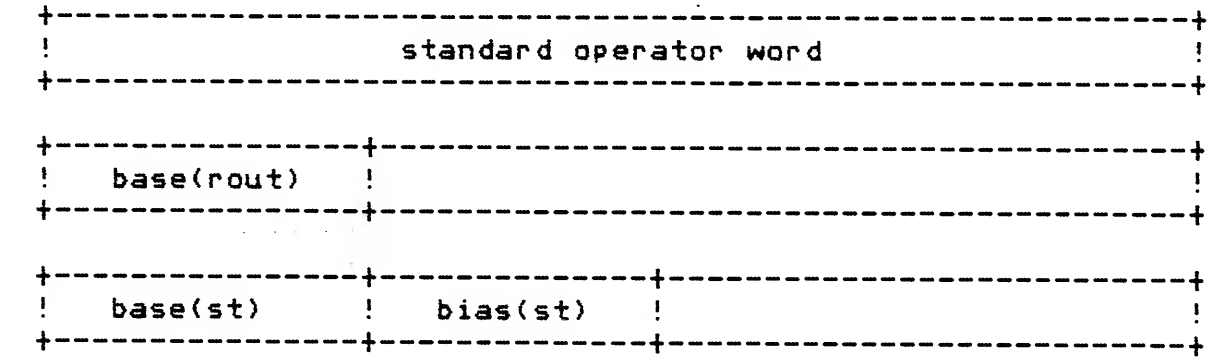
Get_Passed_Length_(GPL)_Tuple

This tuple has the passed length of a type character dummy argument or character function as a result.



Where: C is the symbol table ordinal of the dummy argument or of CVAL. for a character function.

STOP_or_PAUSE_Tuple

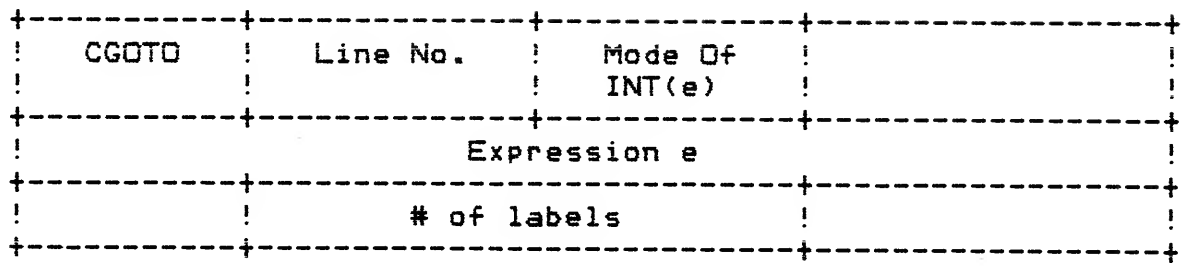


Where: base(st) is CCON. if a string was specified on the STOP statement, or zero.

bias(st) is the offset in the character constant table of the string.

base(rout) is the symbol table ordinal of the entry for the STOP routine name.

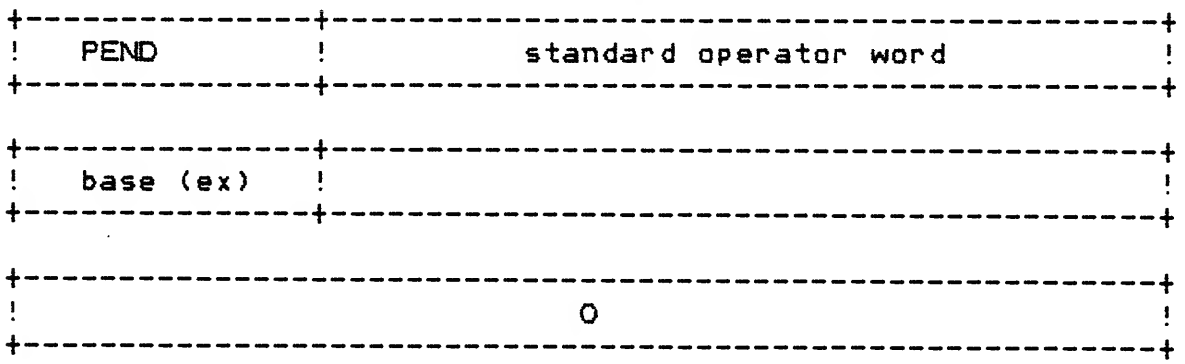
Computed Go To Turple



This turple is produced for GO TO (P1, P2, P3,...,Pn) e. The next n turples following this turple are special unconditional jump turples (JGOTO), one per label in the computed GO TO list.

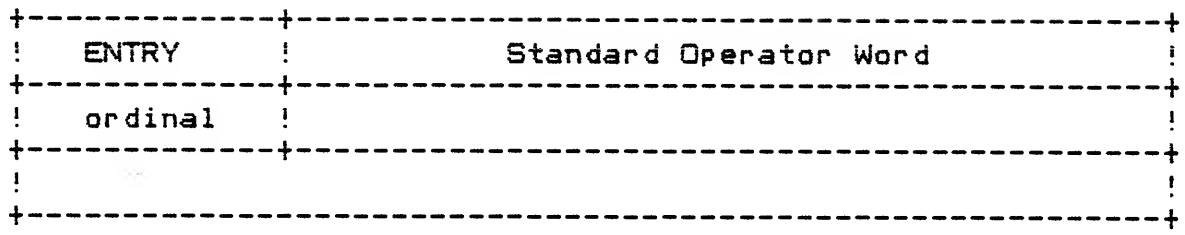
PEND Turple

This turple is output only when control flows into the END line of a main program.



Where: base(ex) is the ordinal of the symbol table entry for the external to be transferred to at program termination. The code from this turple will transfer control to a termination external identified by symbol table entry ex.

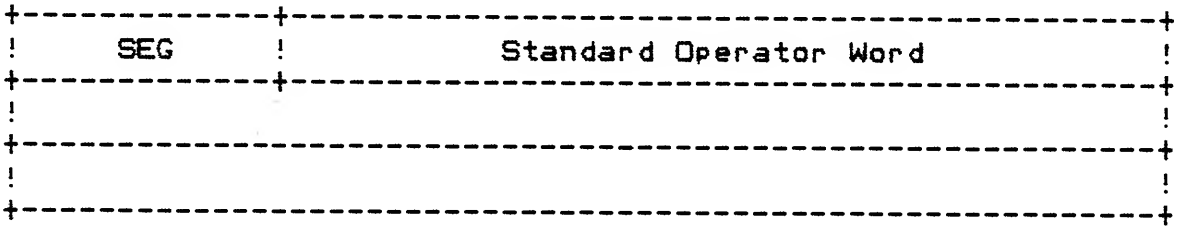
Entry Turple



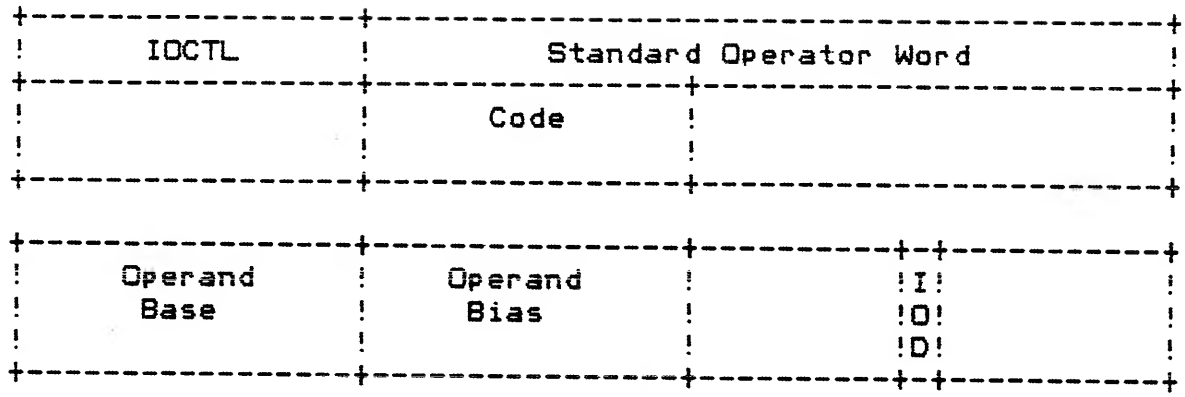
ordinal is the symbol table ordinal of the entry name on the ENTRY statement. A link to he formal parameter table which contains the formal parameter or list for this entry is in the PTRF field of the symbol table entry for the entry name.

SEG Turple

This turple indicates the beginning of a new IL sequence, i.e., the turple numbering is reset to zero, with the turple following the SEG turple being turple number zero.



Control Tuple



I/O routine ordinal is the symbol table ordinal of the I/O routine to be called. The symbol table ordinal is present in the first turple received to communicate the symbol table ordinal and allow usage/definition information collection in OPT=2.

Code is an integer specifying the numeric control code of this turple.

UNIT=	1
END=	2
ERR=	3
FMT=	4
IOS=	5
REC=	6
SKIP	7
NML	8
ACCESS=	12
BLANK=	13
BUFL=	14
DIRECT=	15
EXIST=	16
FILE=	17
FORM=	18
FORMATTED	19
NAME=	20
NAMED=	21
NEXTREC=	22
NUMBER=	23
OPENED=	24
RECL=	25
SEQUENTIAL=	26
STATUS=	27
UNFORMATTED=	28
BUR	29
CNT	30
MOD	31
STR	32
FMTA	33

BUFFER I/O fwa/lwa, parity and ENCODE/DECODE string, length are also implemented as control turples.

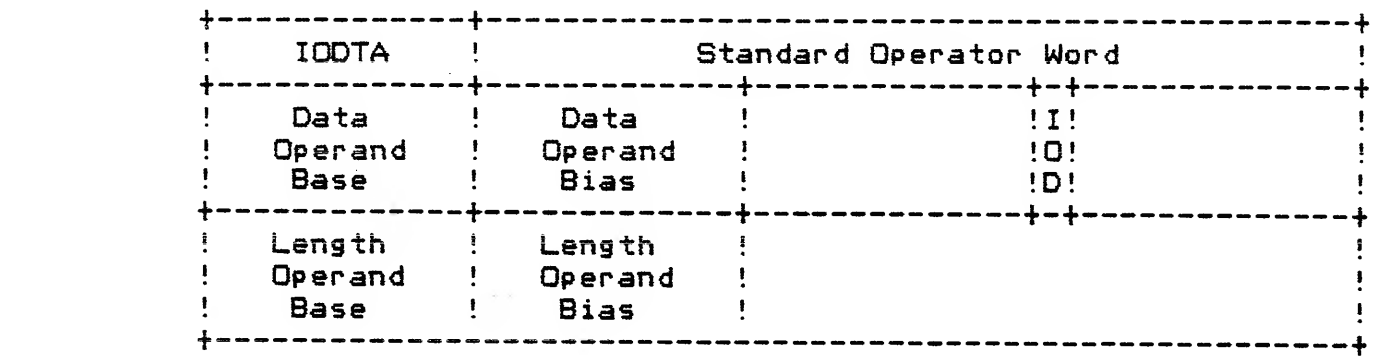
Operand_Base/Operand_Bias designate the operand of the control turple (i.e., REC=X specifies X, ERR=10 specifies label 10, etc.)

Ordering_of_Control_Turples

The turples will appear inthe order in which their corresponding specifiers are encountered in the source.

Data Tuple

All IODTA turples for a segment of an I/O list will immediately precede the IOF call that processes that segment.



Data operand Base/Bias specifies the data item to be read into or written from.

Length operand Base/Bias specifies the length of the data item. A constant length is contained in the bias field and the base field is zero. Constant lengths may exceed the length of a short constant bu are still indicated by the short constant bit being on. In all cases, lengths are in terms of number of items. A double precision, complex, or character variable is of length one. (Character element length is available in the Symbol Table.)

Non-collapsible I/O DO Loops

Non-collapsible I/O DO's will result in a normal DO control structure surrounding one or more restart calls to perform the data transfer.

IOF Tuple

Operand 1 specifies the symbol table ordinal of the routine to be called.

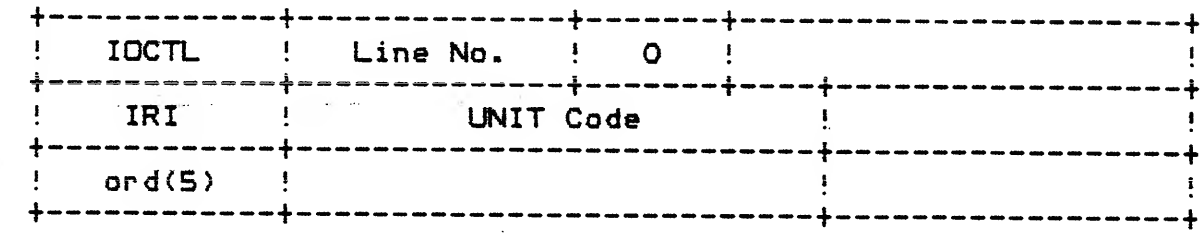
Operand 2 specifies the restart routine symbol table ordinal or zero if this is the last call of the I/O statement. Restart calls are required for situations such as READ (1) I, A(I).

I/O Tuple Examples

In the following example, I is an integer variable, D is a double precision variable, and A and B are real arrays with a lower bound of one.

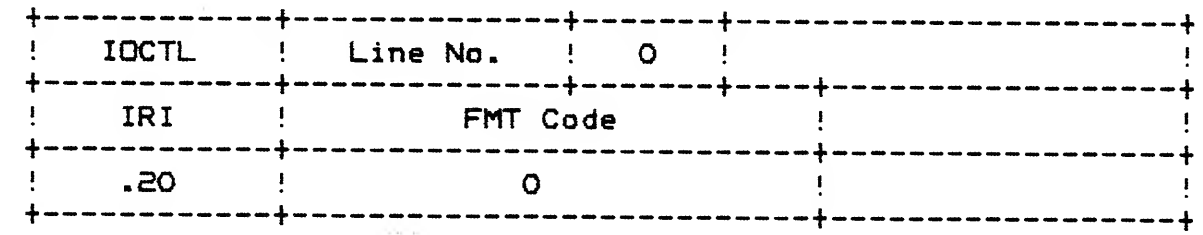
```
READ (5,20,REC=N,END=10) I,D,A(I),(B(J),J=1,10)
```

would be represented as



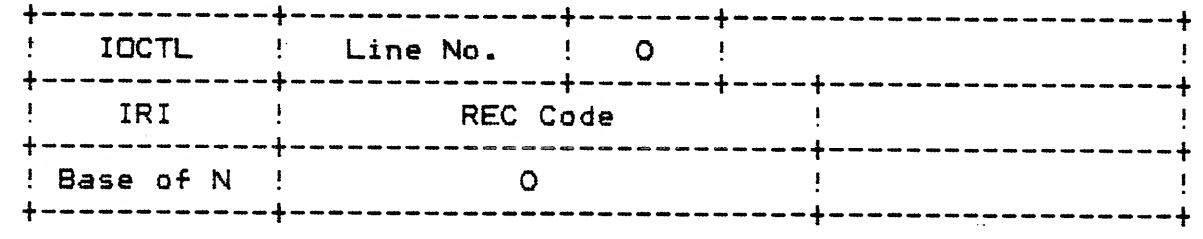
IRI = symbol table ordinal of input routine for formatted direct access input initial call.

UNIT code = numeric code for UNIT= specifier.



FMT code = numeric code for FMT= specifier.

.20 = symbol table ordinal of entry for label 20.



REC code = numeric code for REC= specifier.

IOCTL	Line No.	0	
IRI	END Code		
.10	0		

END code = numeric code for END= specifier.

IODTA	Line No.	Mode				
Base of I	0					
0	1		S C		I O D	

Mode is integer for I, the length is one.

IODTA	Line No.	Mode				
Base of D	0					
0	1		S C		I O D	

Mode of D is double precision.

IOF	Line No.	0	
IRI	0		
IRR	0		

Call the input routine to issue part of the I/O list. Restart call generated due to interaction of I and A(I).
IRI = symbol table ordinal of routine to call.
IRR = symbol table ordinal of input restart routine.

SUBSC	Line No.	Mode		PAP	FIP	
Base of A	-1					
Base of I	0					

SUBSC = subscript turple
Mode is real for A

IODTA	Line No.	Mode						
PTR	0		INT		ARY			
0	1			SC			IOD	

PTR points to the previous turple in the IL.
INT designates an intermediate, i.e. ordinal field is a pointer to a previous turple.

IODTA	Line No.	Mode					
Base of B	-1						
0	10			SC		IOD	

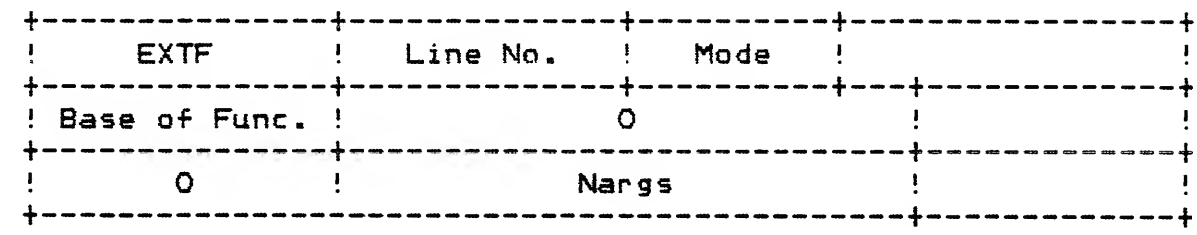
I/O List DO collapse transforms (B(J),J=1,10) into a transfer starting with B(1) and transferring 10 items.

IOF	Line No.	0				
IRR	0					
0	0					

Issue last restart call

External Routine Reference Turple

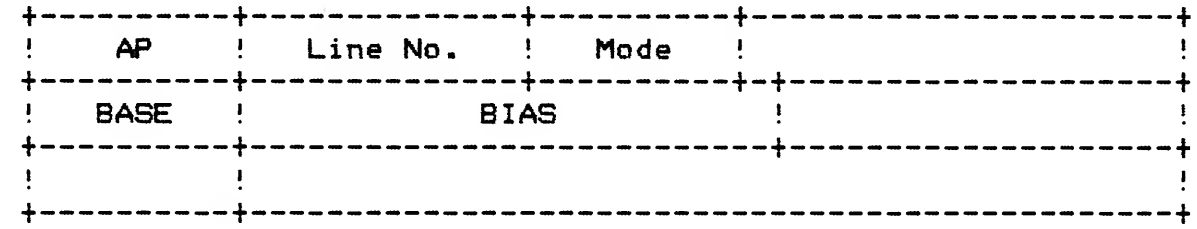
The general form of an external function reference (CALL or function) turple is:



Mode is the mode of the function result.
Base of Func. is the symbol table ordinal of the function or subroutine name to be called.
Nargs is the number of arguments it was referenced with.

Actual Parameter (AP) Turple

This turple specifies an actual argument of a subsequent subroutine or function call. All AP turples will immediately precede the associated EXTF turple. The form is:



Where: BASE is the symbol table ordinal of the actual parameter and BIAS is the offset.

ARG_Turple

This turple is identical to the AP turple except that it denotes actual arguments to intrinsic functions.

Example

In the following example, F is a function, A, Y, Z, R and I are variables, and B is an array with lower bound of one.

```
CALL X(A, Y+Z, F(R), B(I))
```

R.ADD	Line No.	Mode	P A P F	
Base of Y	Bias of Y			
Base of Z	Bias of Z			

Compute Y+Z

AP	Line No.	Mode	
Base of R	0		
	0		

Aplist turple for argument R to function F.

EXTF	Line No.	Mode	P A P	
Base of F	0			
0	1			

Call to function F with one argument.

SUBSC	Line No.	Mode		PAP
Base of B	-1			
Base of I	0			

Compute subscript for B(I).

AP	Line No.	Mode	
Base of A	0		
0	0		

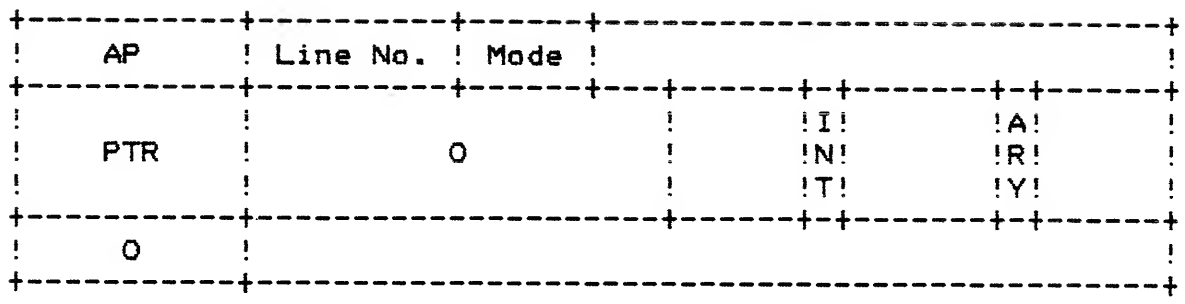
First argument of CALL.

AP	Line No.	Mode		TNT
PTR	0			
0				

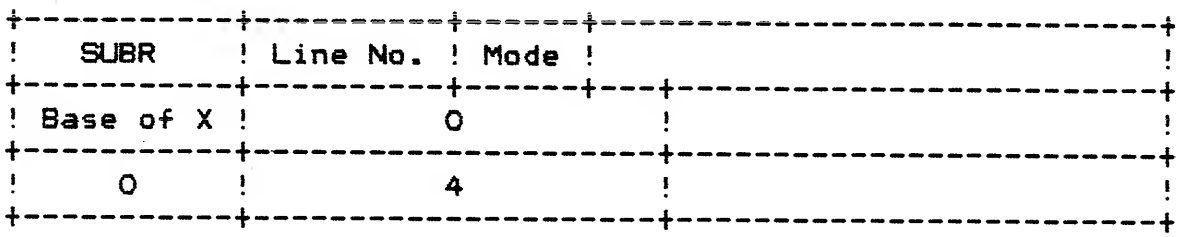
PTR points to the Y+Z turple. This is the second argument turple.

AP	Line No.	Mode		TNT
PTR	0			
0				

PTR points to the EXTF turple of the call to F. This is the third argument.



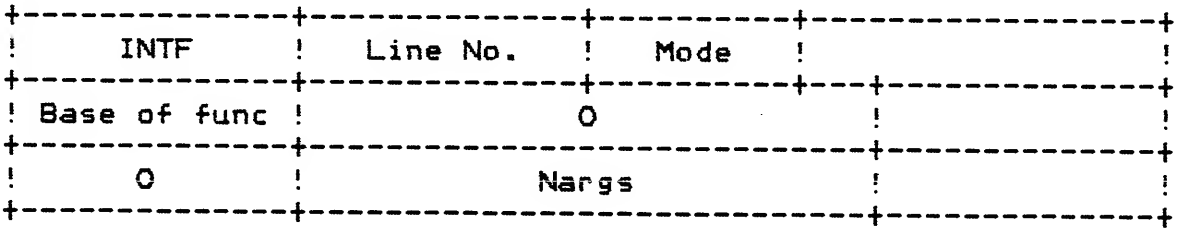
PTR points to the B(I) turple. This is the fourth argument.



Call to subroutine X with four arguments.

Intrinsic Function Turples

Inline intrinsic functions will be generated by the front end as explicit turples to perform the required operations. Out-of-line intrinsic functions will generate ARG turples followed by an INTF, intrinsic function call turple.



Mode is the mode of the function result.
Base_of_func is the symbol table ordinal of the internal function name (SIN., SQRT#, etc.) to be called.
Nargs is the number of arguments.

2.1.8 Namelist Group Table (T.NLST)

A variable length table (depending on the number of members in the group).

NG.	NMEM	GROP	MEM1	MEM2
	15	15	15	15

NMEM: Number of members in the namelist group
GROP: Symbol table ordinal of group name
MEMn: Symbol table ordinal of nth member

Subsequent entries are as MEM1, MEM2, 4 per word.

2.1.9 Actual Parameter List Table (T.APL)(T.IOA)

Each AP list consists of one entry per item.

```
      +--------------+----------------------+------+------+----+
IA.   !      TAG     !         BIAS         ! MODE ! ATTR !////!
      +--------------+----------------------+------+------+----+
             18                 24             6      7     5
```

TAG: T.PB form of *IH* = symbol table ordinal
BIAS: Constant offset
MODE: Mode (type)
- IOC: Control item
- ST: FWA stored to this item
- CHAR: Get (BCP, CLEN, BIAS) from T.CAC
- CRH: Character relational header
- ASG: Assigned format specifier
- FP: Formal parameter
- VAR: Variable do trip indicator

The T.IOA entries are reformatted as follows:

Noncharacter/FP

```
      +--------+-------+--------------+-------------------------+
0A.   !  ATTR  !  TYP  !      LEN     !          ADDR           !
      +--------+-------+--------------+-------------------------+
           6       6          18                 30
```

Character

```
      +--------+-------+--------------+-+-+----------------------+
      !                               !/!B!                      !
0A.   !  ATTR  !  TYP  !      LEN      !/!C!         ADR          !
      !                               !/!P!                      !
      +--------+-------+--------------+-+-+----------------------+
           6       6          18       2 4          24
```

FP

```
      +------+-----+--------------+--------------------+---------+
0A.   ! ATTR ! TYP !      LEN     !        SUBS        !   ARG   !
      +------+-----+--------------+--------------------+---------+
          6     6          18               21             9
```

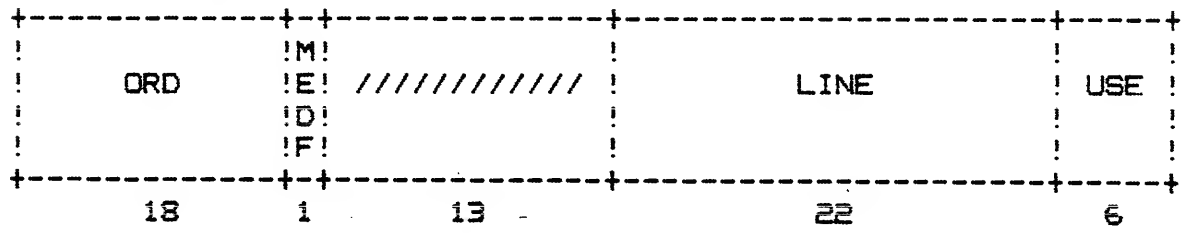
ATTR:
- LCM: In ECS/LCM
- FP: Formal parameter
- IND: Indirect
- LST: Control information specifier
- VAR: Variable do trip indicator

TYP: Mode or unit control code
LEN: Number of elements
ADDR: FWA
BLP: Beginning character position
ADR: FWA
SUBS: Offset to formal parameters
ARG: Formal parameter index

2.1.10 Cross Reference Table (T.REF)

D16

One entry per symbol reference. Gathered (as requested) during front end processing and processing during rear end listing production.



ORD: Symbol table ordinal
MEDF: Map entry point definition
LINE: Line number of reference
USE: Usage symbol

2.1.11 Entry Point Table (T.ENT)

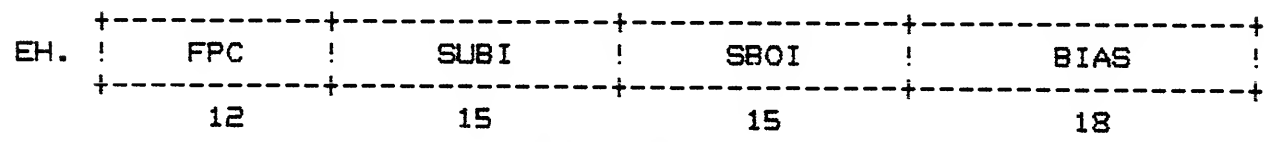
One entry for each entry point in a program unit.

NAME	ORD
42	18

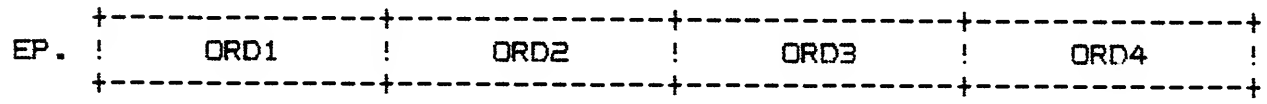
NAME: Entry point name (DPC, left justified, zero filled)
ORD: Symbol table ordinal

2.1.12 Entry Point Parameter List Table (T.ENTP)

Each unique parameter list in a program unit has a T.ENTP entry. An entry consists of a header (EH.) and as many EP. words as necessary. A list is terminated by one (or more) zero bytes.



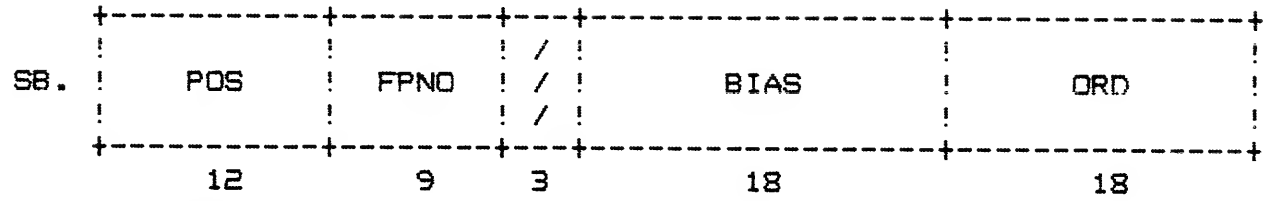
FPC: Formal parameter count
SUBI: Subindex table bias
SBOI: Level 0 subindex table bias
BIAS: CPL. bias of this list



ORDn: Symbol table ordinal of the nth FP

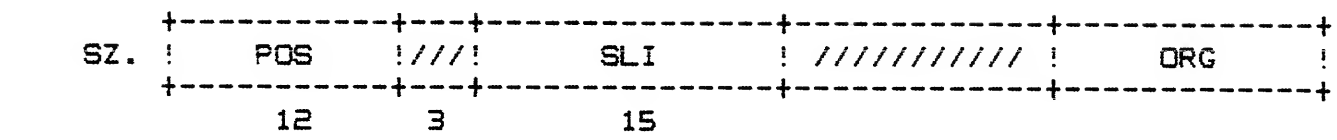
2.1.13 Sub Block Table (T.SUB)

E3



POS: Instruction parcel shift count
FPNO: Formal parameter number
BIAS: Bias added
ORD: Address of instruction to address substitute

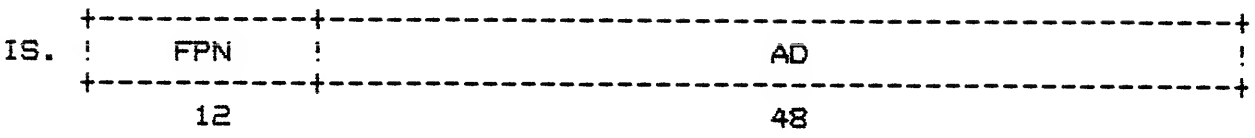
2.1.14 SubO Block Table (T.SUBO)



POS: 2036B + instruction parcel shift count
SLI: SCM/LCM store instruction
ORD: Address of instruction to address substitute

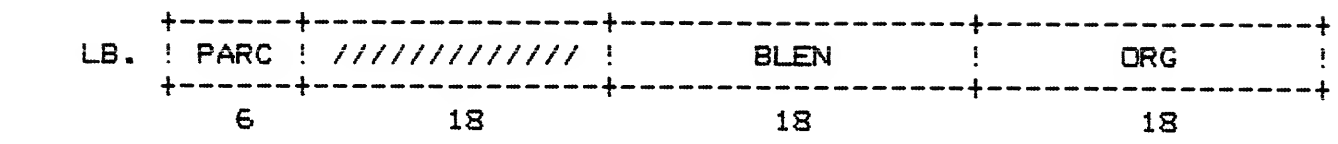
2.1.15 Sub Block Index Table (T.SBI)

E5



FPN: Formal parameter number
AD: Address of Sub, this FP

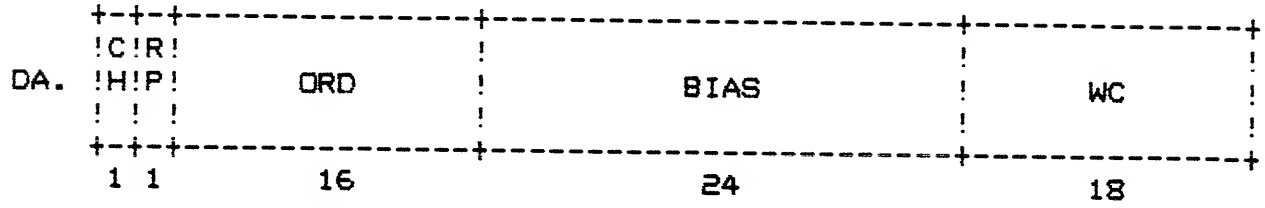
2.1.16 Local Block Table (fixed table)



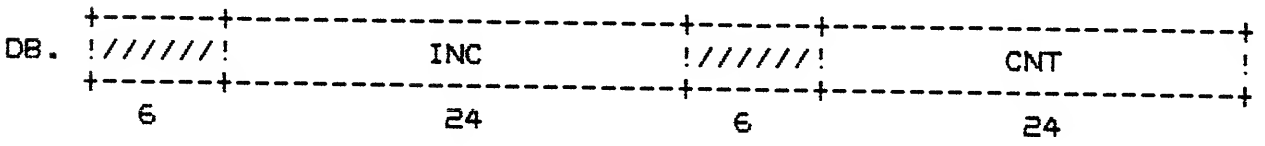
PARC: Parcel count
BLEN: Length of block
ORG: Program relative address/org counter

2.1.17 Data Statement Table (T.DATS)

The entries consist of a one- or two-word header and as many constant entries as necessary.



CH: Character format
RP: Replication needed (DB. word present)
ORD: Symbol table format of FWA
BIAS: Bias off FWA
WC: Count (words or characters) not including DB.

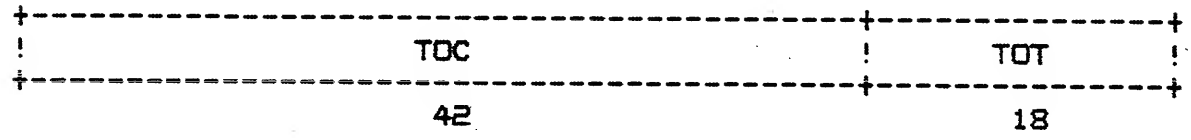


INC: Increment
CNT: Number of copies

2.2.1 Token Buffer (TB.)

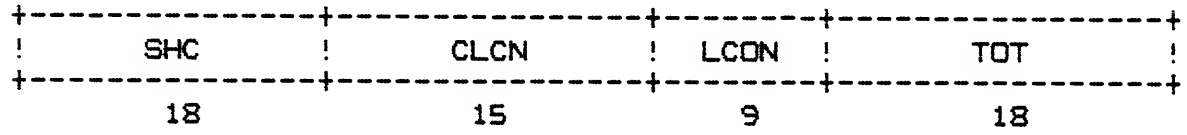
The token buffer is the output of LEX (lexical analysis). LEX entokens an entire statement (or C$ directive) into T.TB. T.TB is always at a fixed location and its FWA doesn't move as long as the token buffer is active. Token formats are as follows:

Normal Tokens



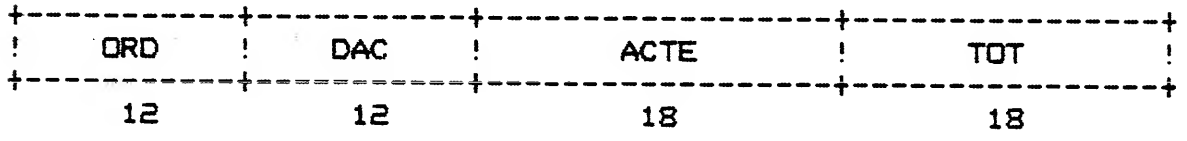
TOC: Token character string (variable and constant)
TOT: Token type (value)

LEX Formed Constant Tokens (Hollerith, Character, Octal)



SHC: Pointer to constant in constant table
CLCN: Length of constant (Hollerith and character)
LCON: Number of words in constant (Hollerith and character)
TOT: Token type (value)

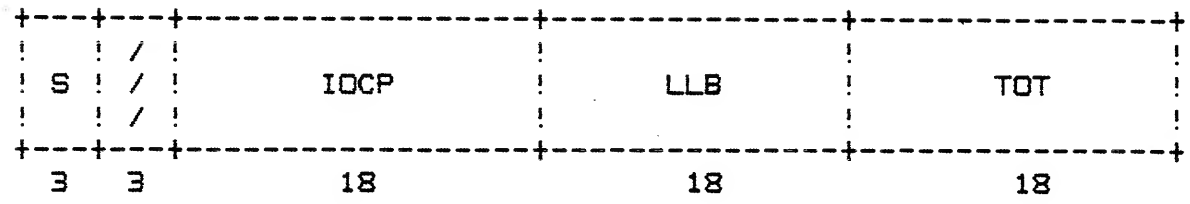
Statement Function Dummy Argument (made by STMTF)



ORD: symbol table ordinal of dummy argument
DAC: Dummy argument reference chain pointer
ACTE: Actual parameter address (reference time)
TOT: Token type = O.STFA

Parentheses Tokens: These tokens originally are output by LEX and may be modified by the implied do list processing.

Left Parenthesis (LEX)

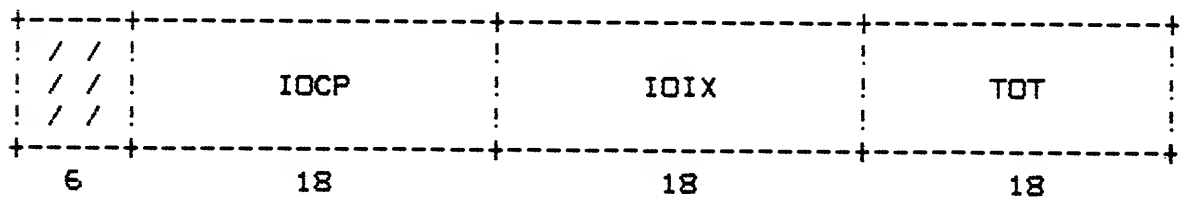


S: Switches
COL: Paren level contains a colon
EQL: Paren level contains an equal sign
SBS: Array substring

IOCP: Pointer to matching right paren
LLB: Pointer to next outer left paren (or null)
TOT: Token type (value = 0.LP)

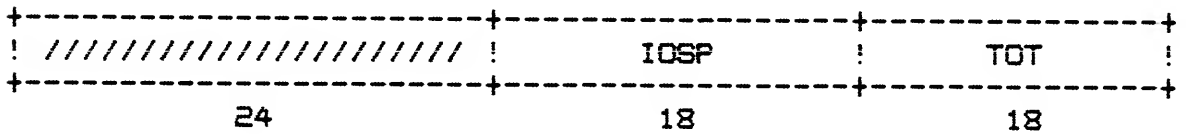
Right Parentheses (LEX): Normal token format

Left Parenthesis (DO begin)(IO)



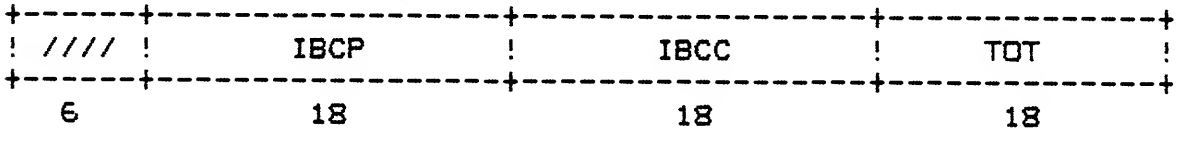
IOCP: Pointer to matching right paren
IOIX: Pointer to implied loop index
TOT: Token type (value=0.DOBI)

Right Parenthesis (Do conclusion)(IO)



IOSP: Implied loop and marker (for parser)
TOT: Token type (value=0.DOCI)

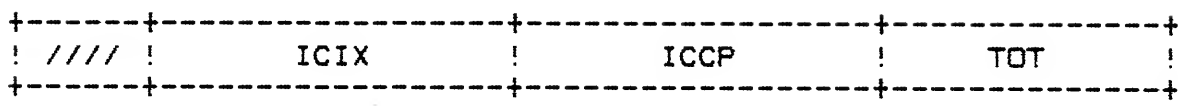
Left Parenthesis (Do collapse begin)(IO)



IBCP: Pointer to closing right paren
IBCC: Pointer to collapse conclusion token
TOT: Token type (value=0.DCBI)

Do Collapse Conclusion (IO)

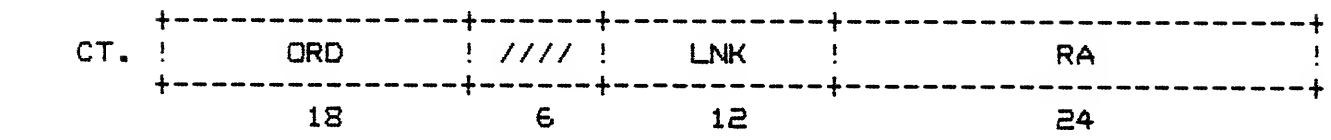
E10



ICIX: Symbol table ordinal of Do index
ICCP: Pointer to closing right paren
TOT: Token type (value=O.DCCI)

2.2.2 Common Item Table (T.COMM)

The common item table contains one (one word) entry for each variable that appears in a common declaration.

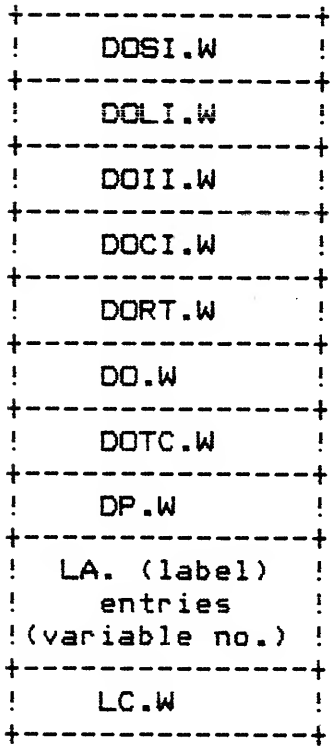


ORD: Symbol table ordinal of variable
LNK: Link to next member in same block
RA: Relative address within the block

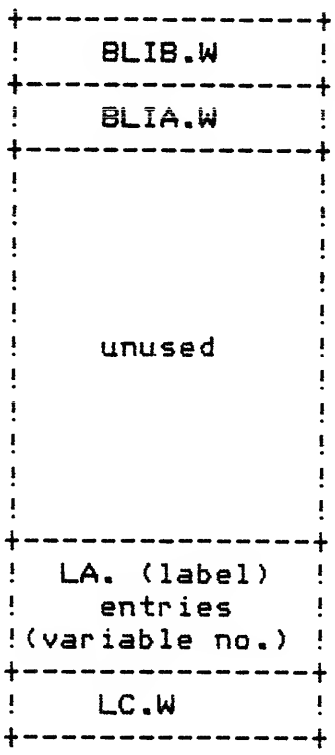
2.2.3 Block Structure Table (T.BLST) E12

The block structure table contains a variable length entry for each active program structure (Do loop or block IF) while the structure is still active. The block structure table entry format is as below:

For DO loop



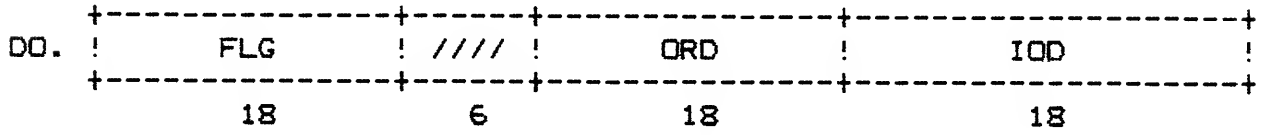
For block IF



For individual cells, the following formats apply:

DOSI.W: Do loop initial value (TP. format)
DOLI.W: Do loop limit value (TP. format)
DOII.W: Do loop increment value (TP. format)
DOCI.W: Do loop control index (TP. format)
DORT.W: Inverted label of loop top (TP. format)
DO.W: Do loop terminating label

DO.

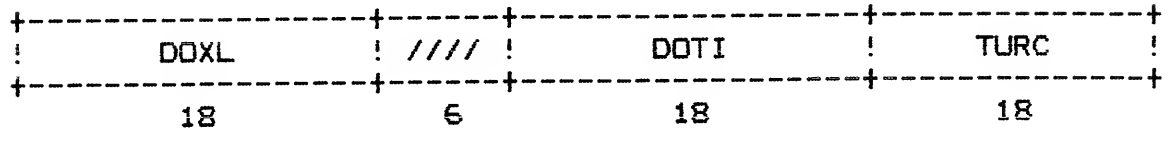


FLG: Do begin turple ordinal (T.PAR)
ORD: Symbol table ordinal of terminal label
IOD: Nonzero if implied do

DOTC.W: Compiler generated trip count variable (TP. format)

DP.W: Do loop information

DP.



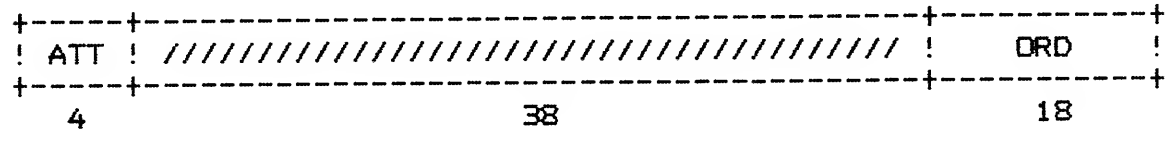
DOXL: Generated ordinal of do exit (if zero trip)
DOTI: Symbol table ordinal of trip count variable
TURC: Do conclusion skeleton ordinal

BLIB.W: Invented label of block bottom (TP. format)

BLIA.W: Invented label of next false branch (TP. format)

LA.W: Structure label encountered

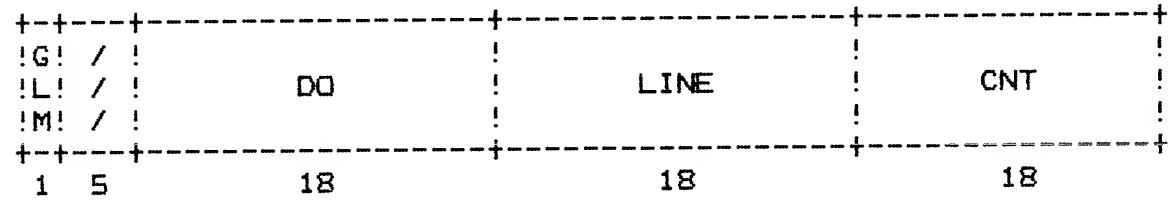
LA.



ATT: Attributes
- DEF: Label defined in structure
- REF: Label referenced in structure
- EXT: Label is exit from a do loop
- ENT: Label is entry in do loop

ORD: Symbol table ordinal of label

LC.W: Structure count word



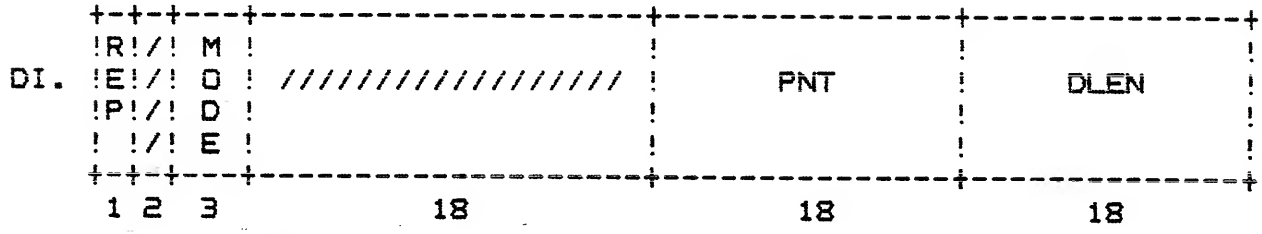
GLM: Generated bottom label must materialize
DO: Symbol table index of header label (Do loop)
LINE: Block origin source line
CNT: Number of words in this block structure entry

2.2.4 Data Constant Table (T.DATI)

E14

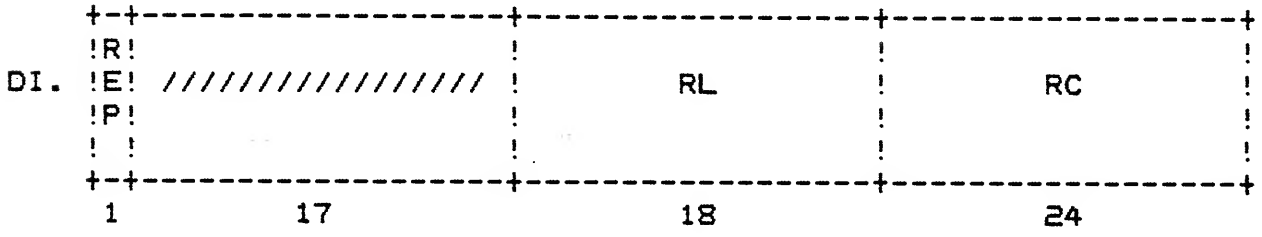
Built when process DATA constant list. There are two forms of entry:

Data Constant



REP: Replication (=0)
MODE: Constant mode
PNT: Pointer to constant
DLEN: Length (words or characters)

Replication Header

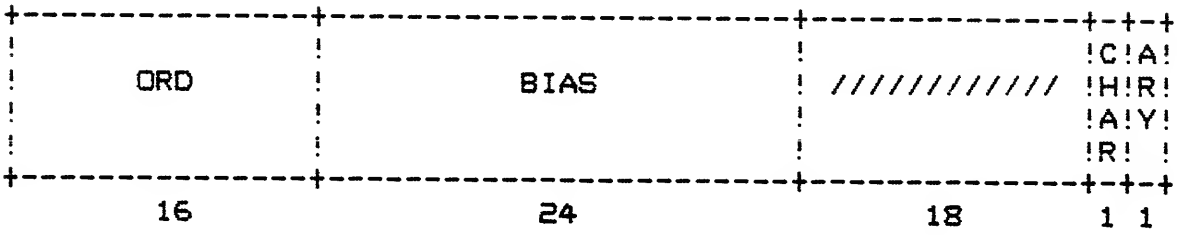


REP: Replication (=1)
RL: Repetition list length
RC: Replication count

2.2.5 Input List Item Table (T.ILI)

E15

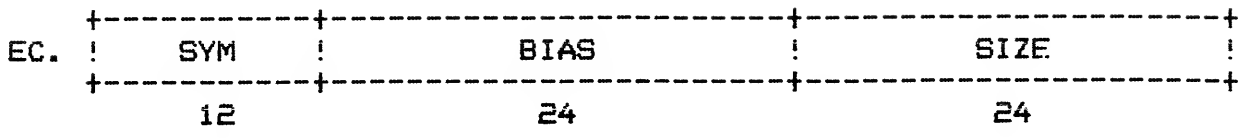
The information in this table is gathered to decide upon issuing restart I/O calls on subscripted variables.



ORD: Symbol table ordinal
BIAS: Bias (constant offset)
CHAR: Operand is type character
ARY: Operand is indexed array or substring

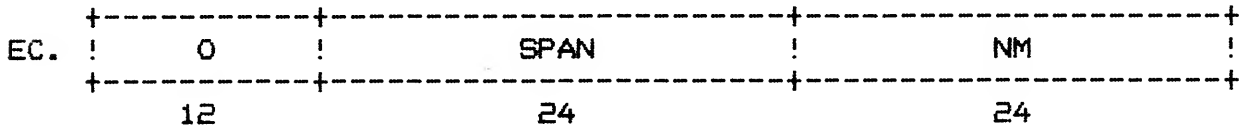
2.2.6 Equivalence Class Table (T.ECT)

Class Member



SYM: Symbol table ordinal
BIAS: Offset of member from class base
SIZE: Length of member

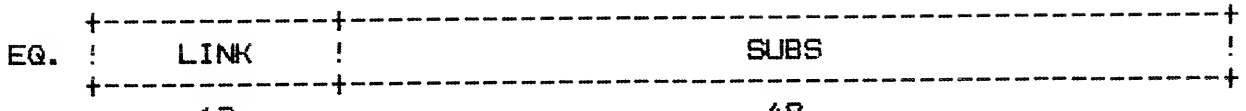
Header



FIRST 12 bits zero
SPAN: Length of class
NM: Number of members in class

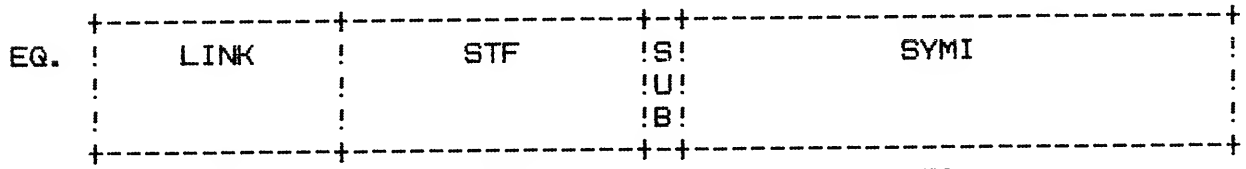
2.2.7 Equivalence Name Table (T.EQUS)

Noncharacter



LINK: Index of member
SUBS: Subscript of this item

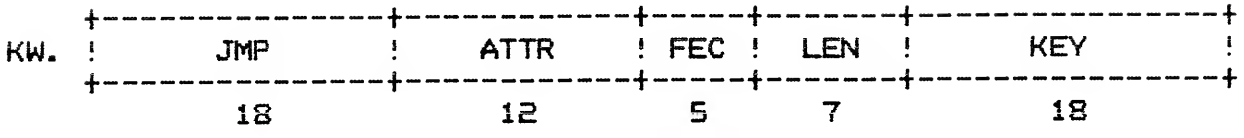
Character



LINK: As above
STF: Substring first
SUB: Item subscripted
SYMI: WB.W index of member

2.2.8 Keyword Table (fixed table)

Used by front end processor FEC/LEX.



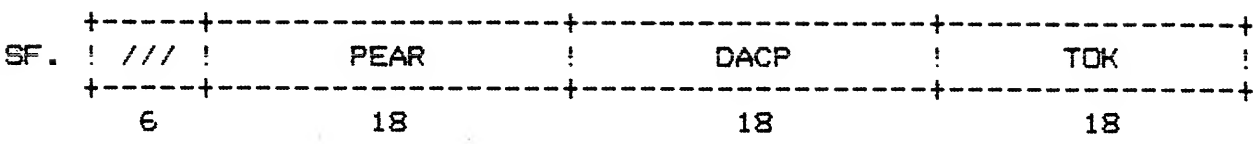
JMP: Address of the relevant statement processor
ATTR: Attributes (see below)
FEC: Context legality of statement (position)
LEN: Length of keyword
KEY: Address of keyword literal string

KW.ATTR

- DON: May not be Do terminal statement
- NIF: May not be object of logical if
- LBL: Statement may have referencable label
- GEN: Statement generates turples
- BKD: Statement is legal in Block Data
- PWS: Statement is to be processed in skip mode
- IL: Statement has implied label
- NBS: Generates turples but no BOS is output

2.2.9 Statement Function Header (T.STF)

F3

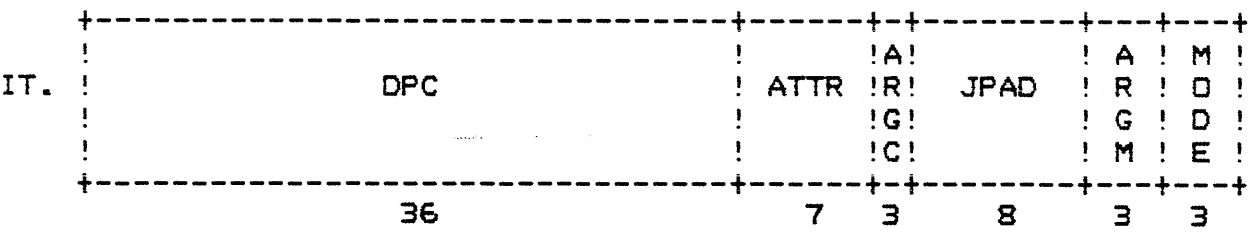


PEAR: Previous ESTACK base for actual arguments
DACP: Dummy argument reference chain head
TOK: Dummy token (for PAR)

2.2.10 Intrinsic Function Table (fixed table)

F4

One entry per intrinsic function (both inline and external)

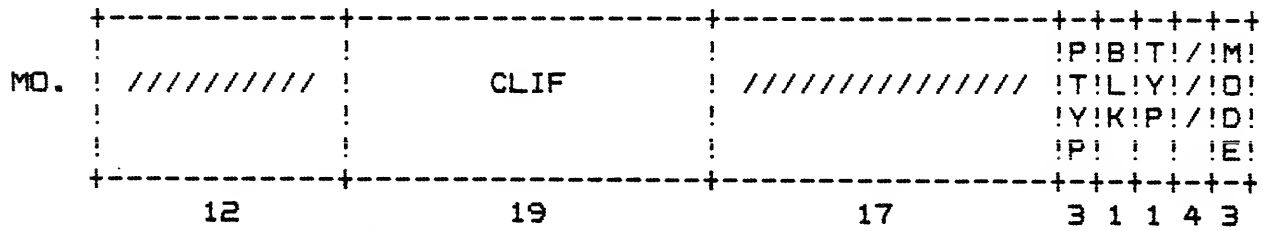


DPC: Intrinsic name (DPC, left justified, zero filled)
ATTR: Attributes (see below)
ARGC: Required number of arguments
JPAD: Location of function description
ARGM: Mode of arguments
MODE: Mode of result

IT.ATTR
CHAR: Character function
BYN: Always call by name
GENF: Generic name
XTER: External intrinsic
ANSI: Defined by ANSI
PAR: Parser has special processing

2.3.1 Program Unit Mode (MOD)

The cell MOD is used by the front end processor to determine legality of syntax and semantics.



CLIF: Same as WC.CLIF character information
PTYP: Program unit type
FUN: Function
SUB: Subroutine
PRO: Program
BLK: Block Data
TYP: Explicitly typed function
MODE: Mode (function only)

2.3.2 Parse Control Cells (ARGMODE, ARGCOMA, ARGMISC)

F6

These cells control end of expression transition in the parser.

ARGMODE

AM.

REF	ATR	COM	PAD
12	12	18	18

REF: Cross reference symbol at current time
ATR: Attributes
- ARE: Allow unsubscripted array name
- LEV3: Allow level 3 name
- COL: Allow colon
- EQ: Allow =
- RP: Special right parenthesis processing
- EOS: Allow end of statement to unstack LP
- FUN: Allow function reference without list

COM: Address of routine to process comma delimited end of expression
PAD: Address of routine to process right parenthesis delimited end of expression

ARGCOMA

Array Subscript Processing

AC.

V / S / U / B	SYM	DIMI	CNT
1 5	18	18	18

VSUB: Subscript constant flag
SYM: Symbol table ordinal of array
DIMI: Dimension table index
CNT: Subscript expression count

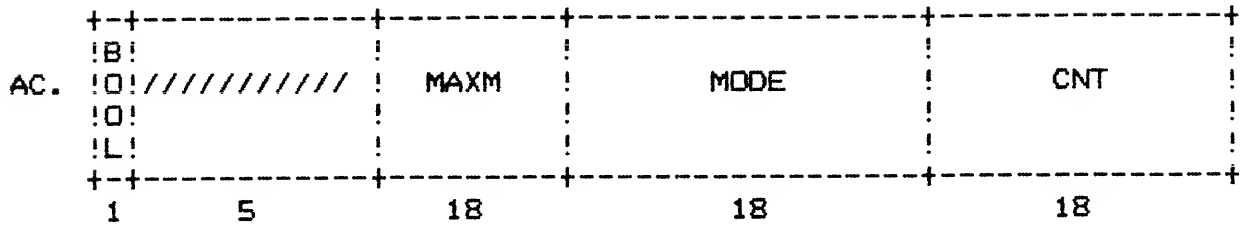
Call or Function Argument List

AC.

//////////////////////	MODE	CNT
24	18	18

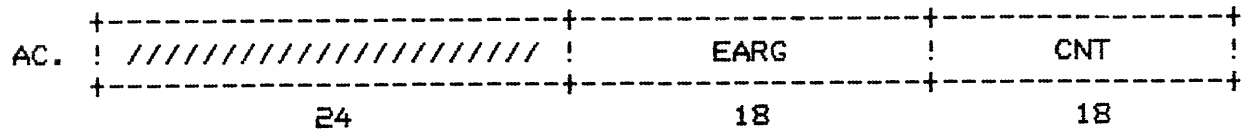
MODE: Mode (function only)
CNT: Arguments processed (-1)

Intrinsic Function Argument List



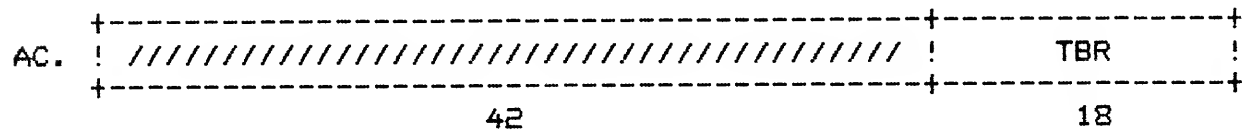
BOOL: Indicates Boolean argument occurred
MAXM: Maximum argument mode
MODE: Intrinsic mode
CNT: Arguments processed (-1)

Statement Function Arguments



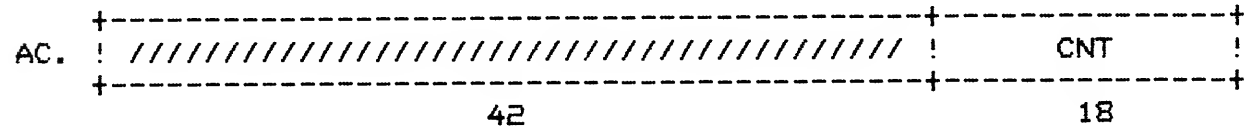
EARG: Address of actual argument (on ESTACK)
CNT: Count of arguments processed

Statement Function Body



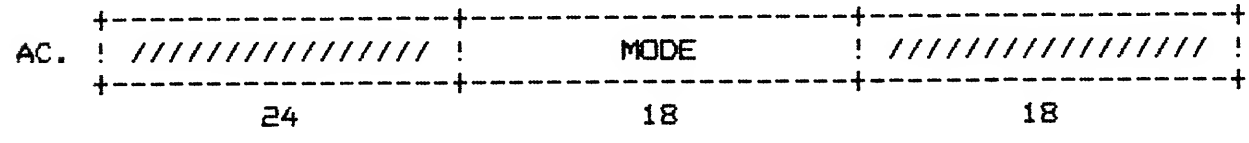
TBR: B4 restore address

Do Loop Indices



CNT: Do parameter count

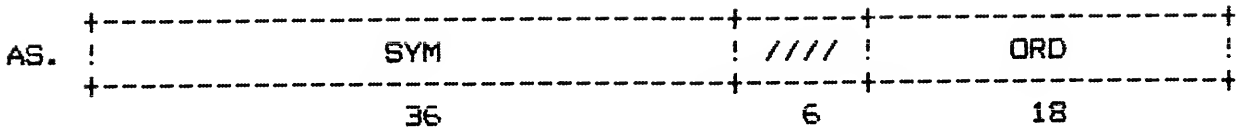
Character Substring



MODE: Mode of variable

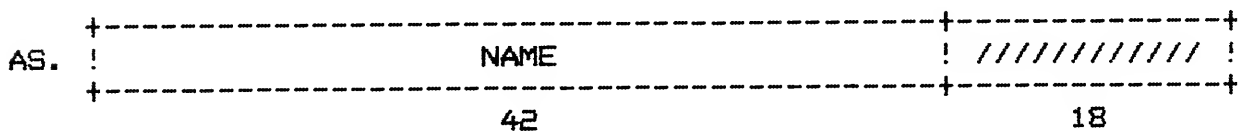
ARGMISC

Basic Intrinsic Function



SYM: Intrinsic function name (DPC, left justified, zero filled)
ORD: Symbol table ordinal

Array Subscripts



NAME: Array name (DPC, left justified, zero filled)

COMCDXB	UTILITY, IDP	DPC to binary conversion routine DXB	External
COMCIDP	IDP	Interactive Debug Package	External
COMCLFM	IDP	Local file manager	External
COMCMCS	IDP	Collate routine MCS	Internal
COMCMNS	UTILITY	Move nonoverlapping bit string - MNS	External
COMCMVE	UTILITY	Move routine MVE	External
COMCPAC	INITOO	Control statement processor	Internal
COMCRDC	UTILITY, IDP	CIO coded input routine	External
COMCRDW	UTILITY, IDP	CIO buffered input routines	External
COMCRSR	IDP	Register restore routine	External
COMCSBM	UTILITY, IDP	Set block of memory to supplied value	Internal
COMCSFN	UTILITY, IDP	Space fill name routine	External
COMCSST	UTILITY	Shell sort routine	External
COMCSTF	INITOO	Set terminal file routine	External
COMCSVR	IDP	Register save routine	External
COMCSYS	IDP	Process system request	External
COMCTOK	IDP, LEX	Token generation routines	External
COMCWOD	UTILITY, IDP	Octal DPC conversion	External
COMCWTC	IDP	CIO coded output routine	External
COMCWTH	UTILITY	CIO Hollerith output routine	External
COMCWTO	UTILITY	CIO write one word	External
COMCWTW	UTILITY, IDP	CIO buffered output routines	External

3.0 COMDECKS

FTN5 uses several comdecks. The purpose of comdecks is:

- Standardize functions across the common product line and within a product (FTN5).
- Reduce maintenance (only fix one place).

FTN5 uses two classes of comdecks:

- Internal: On FTN5PL, maintained by the project
- External: On a secondary PL, maintained by owner of secondary PL.

COMDECK	CALLING DECK(S)	CONTENTS	STATUS
CCOMRPV	UTILITY	Reprieve routines RPV, FRA	Internal
COMACPU	FTN5TXT	General CPU macros	Internal
COMADEF	FTN5TXT	Structured field definition macros	Internal
COMAERR	FERRS, RERRS	ERROR macro	Internal
COMAIDP	FTN5TXT	Debug package macros	External
COMAMGM	FTN5TXT	General macros	Internal
COMAQCG	QCGC, FUN, REG, GEN	QCG macros	Internal
COMATOK	IDP, LEX	Token generation macros	External
COMCBUB	IDP, LEX	Burst routine BUB	External
COMCBUN	IDP, LEX	Burst routine BUN	External
COMCCDD	UTILITY, IDP	DPL conversion routine CDD	External
COMCCFD	PUC	Floating DPC conversion routine CFD	External
COMCCIO	UTILITY, IDP	CIO I/O routines	External
COMCCOD	FTN, IDP	Octal to DPC conversion routine COD	External

COMCXJR	IDP	XJR restore code	External
COMCZTB	UTILITY, IDP	Zero to blank conversion	External
COMDDMT	CSNAP, FSNAP, BSNAP	Table dump routines	Internal
COMDTOK	FSNAP	Token buffer dump format	External
COMFCIP	FTN, INIT00, INIT20, OVL10, OVL20	Compiler installation parameters	Internal
COMFDST	DECL	Double word table sort	Internal
COMFECB	PUC	Evaluate constant bias and substring	Internal
COMFFEI	INIT00, INIT10, INIT21	Front end initialization	Internal
COMFERR	FERRS, RERRS	Common error texts	Internal
COMFGFD	GEN, GRIDGE	Generate file definitions	Internal
COMFGOI	INIT00, INIT10	Global overlay initialization	Internal
COMFICP	GEN, BRIDGE	Issue CP and GPL tables	Internal
COMFISA	GEN, CCGC	Issue save A0 on RJ	Internal
COMFITS	QCGC, CCGC	Issue temporary storage	Internal
COMFMAV	GEN, CCGC	Mark vardim appropriate	Internal
COMFOSC	GEN, CCGC	Output addsub code	Internal
COMFPLI	GEN, BRIDGE	Print limit generator	Internal
COMFROR	INIT00, INIT10, INIT21, INIT22	Reset rounded operations	Internal
COMFSCB	FUN, BRIDGE	Subsume constant char bias	Internal
COMFSCS	FEC, RLINK	Scan table with supplied mask	Internal
COMFSID	BRIDGE	Select integer divide skeleton	Internal

COMFSIM	BRIDGE	Select integer multiply skeleton	Internal
COMFSKL	QSKEL, FSKEL	Front end skeleton formater	Internal
COMFSMO	BRIDGE	Select mode subskeleton	Internal
COMFSMK	BRIDGE	Select mask subskeleton	Internal
COMFSSH	BRIDGE	Select shift subskeleton	Internal
COMFTTL	FTN	Title line template	Internal
COMFUSE	QCGC, FAS, CCGC	Process use pseudo	Internal
COMFWIN	QCGC, CCGC	Write instructions to prebinary	Internal
COMPCOM	FTN	COMPASS interface area	External
COMQSVR	PEM, IDP	Save and restore registers	External
COMSEIS	QSKEL, FSKEL, INITOO, INIT10, INIT21, CONRED, GEN and imbedded in COMFSKL	Skeleton description definitions	Internal
COMSERR	PEM, FERRS, RERRS	Error skeleton definitions	Internal
COMSIDP	IDP, FSNAP, CSNAP, RSNAP	IDP interface text	External
COMSIOC	FTN5TXT, IO, FAS	I/O control codes	Internal
COMSLBT	PUC, REC, LIST	Local block origin table	Internal
COMSPBD	FTN5TXT	Prebinary definitions	Internal
COMSPSU	QCGC, CCGC, FAS, FTN5TXT and imbedded in COMFWIN	Pseudo instruction definitions	Internal
COMSQCG	QCGC, FUN, REC, GEN, FSNAP	QCG macros	Internal
COMSQRF	QSKEL, FSKEL, QCGC, FUN, REG, GEN and imbedded in COMFSKL	QCG register associates	Internal

F12

COMSSYM	FTN5TXT, SYMDEFS	Symbol table definitions also definitions for variable dimension information, formal parameter, common block tables.	Internal
COMSSYC	FERRS	Symbolic representations of symbol table class bits	Internal
COMSTAB	PUC, CCGLINK, CCGC, INIT22	Shared tables	Internal
COMSTAD	PUC, CCGLINK, CCGC, INIT22 and imbedded in CGHCDTD and COMSTAB	Shared tables (variable)	Internal
COMSTAS	PUC, CCGLINK, CCGC, INIT22 and imbedded in CGHCSTD and COMSTAB	Shared tables (static)	Internal
COMSTOK	IDP, FSNAP, LEX	Token generator interface	External
DEFINS	QSKEL, FSKEL, FUN, REG, GEN and imbedded in COMFSKL	Machine opcode definitions	Internal
FSCALE	CONRED	Constant conversion routines	Internal
FA=CLO	UTILITY	RM close routine	External
FA=DEFS	FTN5TXT	RM I/O macros and definitions	External
FA=EOF	UTILITY	RM end of file routine	External
FA=EOR	UTILITY	RM end of record routine	External
FA=FLSH	UTILITY	RM flush holding buffer	External
FA=OPE	UTILITY	RM open routine	External
FA=RDC	UTILITY	RM coded input routine	External
FA=RDW	UTILITY	RM buffered input routine	External
FA=RWX	UTILITY	RM rewind routine	External

FA=SET	UTILITY	RM file initialization routines	External
FA=WTH	UTILITY	RM coded output routine	External
FA=WTW	UTILITY	RM buffered output routine	External
OPRDEFS	RLINK, CSKEL	CCG intermediate language definitions	External
OPTIONS	FTN5TXT	Installation parameters	Internal
PARSKEL	Imbedded in COMFSKL, thus in QSKEL, FSKEL	Parser skeleton tables	Internal
SKEL	QSKEL, FSKEL, CSKEL, imbedded in COMFSKL	Instruction skeleton expansions	Internal
SKOP	QSKEL, FSKEL, CONRED, GEN, BRIDGE, CSKEL imbedded in COMFSKL	Skeleton field definitions	Internal
SKPCONQ	QSKEL, FSKEL, CONRED, QCGC, imbedded in COMFSKL	Field type constants	Internal
SKPSET	QSKEL, FSKEL, CONRED, QCGC, imbedded in COMFSKL	SKPSET macro	Internal

B. DECK AND ROUTINE DESCRIPTIONS

This section describes the decks which comprise FTN5, the routines which make up each deck, the interfaces and data structures involved and any special algorithms or 'tricks' employed. This document doesn't recite code, register usage, called routines, etc. For these implementation details, the reader is referred to the relevant listings.

Organization of the section is:

1. Texts
2. Cradle routines
3. Front end routines
4. QCG routines
5. Rear end routines
6. CCG routines

Outline for the individual decks is:

1. Abstract - the function of the deck.
2. Interfaces - what decks the deck interacts with.
3. Data structures defined/utilized.
4. Routine descriptions.

1.0 IEXIS

FTN5 uses system texts to provide macros, micros and assembly constants to the decks comprising the compiler. To be included in a system text, the macro, micro or constant should have general applicability. If a constant is used in only one deck, it should be included in that deck only. If a data structure is used only by a small number of decks (e.g., QCG only), use of a common deck should be considered.

The system texts defined for FTN5 are:

1. FTN5TXT: The general systems text. Used by front end, rear end, QCG and CCG interface decks.

2. CMPLTXT: CCG interface text. Used by CCG interface routines.

3. CCGTEXT: CCG text. Used by CCG decks.

1.1 FTN5TXT: FORTRAN 5 Assembly/Installation Text

F16

Abstract: FTN5TXT consists of macros and micros used by FTN5 to access data structures, construct data structures, provide system interface and provide product/routine interface: FTN5TXT also provides assembly constants pertaining to data structure definition, compile limits, etc.

Interface: FTN5TXT is used to assemble front end, rear end, QCG and CCG interface decks.

Data Structures

a. Options

Provided by comdeck OPTIONS. The installation parameters which are user modifiable at installation time.

b. General Assembly Constants

These constant values pertain to compiler limits (boundary values). The assembly constants provide symbolic representation of compiler and language constraints and are intended as a maintenance aid. If future requirements change, changing the relevant symbolic constant and reassembly of the relevant deck(s) should effect the change.

c. RM I/O

Provided by comdeck FA=DEFS. The constants and macros used for the CYBER Record Manager I/O interface.

d. Data Structure Definition Macros

Provided by comdeck COMADEF. The macros and micros used to provide symbolic representation of data structures. The macros are:

DESCRIBE: Describes the beginning of the structure. Gives the prefix name and the length of the structure. Names the structure (optional).

DEFINE: Field definition macro. Assigns a symbolic name to the bit position (within the DESCRIBEd structure), length (and a mask bit for some values).

DEQU: Defines an equivalent structure to a previously defined field.

REDEF: Resets the field pointer for redefinition of a field.

BFLIT: Creates a bit field mask literal.

BFMIC: Creates a bit field micro.

BFMW: Creates a bit field mask word.

e. Debug Macros

Test mode only. Macros to provide snap and dump output. Interface to IDP.

CORE: Snapshot of core.
DUMPT: Table dump.
STRING: Token buffer dump.
PRINT: Print contents of core locations.
USF=: Generate user flag parameter cell.

f. IDP Macros

Test mode only. Provided by comdeck COMAIDP. Interactive debug package interface macros.

BREAK: Place a breakpoint.
FRK=: Generate frequency parameter list.
REG: Register snapshot.
RGR=: Generate register parameter list.
SNAP: General snapshot interface.
SNG=: Generate indirect address fields.

g. General CPU Macros

Provided by comdeck COMACPU. Macros of a general nature, suitable for use by any product.

BITMIC: Generate a micro of bit fields.
LETMIC: Generate a micro of bits, corresponding to an alphanumeric string.
BXQ: OPDEF to clear an X register.
IXI
XJ/XK: Integer division OPDEF.
IXI
XJ/XK,
BN: Integer division OPDEF.
MOVE: Move data block.
SUBR: Subroutine entry/exit definition.

h. General Macros Comdeck COMAMGM.

BSSENT: Generate an entry point and BSS.
BSZENT: Generate an entry point and BSSZ.
CALL: Issue RJ to external subroutine.
BC: OPDEF to convert character count to bit count.
CONENT: Generate an entry point and CON.
CW: OPDEF to convert character count to word count.
EQUENT: Generate an entry point and EQU.
EQUEXT: Generate an external and EQU.
HXQ: OPDEF to shift a structure field to the high order bit position.
ISUSE: Issue USE pseudo.
LDBIT: Set one bit in a register.
LDX: Load register with a value.
LXQ: Left shift redefine OPDEF. Eliminates zero shifts.
MOVEB: Move bit string.
MXX+X: OPDEF for maximum function.
MXX-X: OPDEF for minimum function.
RMT=: Force micro evaluation for remotes.
RPVDEF: Entry definition for REPRIEVE.
RPVFWA: FWA definition for REPRIEVE.
RPVON: Turn on REPRIEVE.
RPVOFF: Turn off REPRIEVE.
SBIT: Shift a specified bit into the high order (sign) position.
SETMEM: Set memory block to given value.
WC: OPDEF to convert word count to character count.
WXX: OPDEF to convert character count to word count and remaining character count.

i. Compiler Specific Macros

Macros to provide function relevant only to FTN5.

ACTTAB: Activate a table (managed).
ADDREF: Add a reference to the cross reference table.
ADSYM: Add an entry to the symbol table.
ADDWD: Add a word to a table.
ALLOC: Allocate managed table space for a table.
INATAB: Inactivate a table (managed).
ANSI: Process ANSI diagnostic.
CLAS= Load class bits into X register.
EMIT: Emit a turple.
FATAL: Process FATAL diagnostic.
HEREIF: Define statement processor.
LITKEY: Generate a keyword literal.
LOVER: Load a FTN5 overlay.

PIA: Convert instruction for listing.
PLINE: Coded output interface.
PLUG: Modify compiler code during execution (test mode only).
SCAN: Search routines interface.
SECT: Assembler group 1 instruction generation.
SHRINK: Collapse managed table to given length.
SUBKEY: Define subkeyword.
SYMASK: Generate symbolic mask.
TAGSEX: Tag invented external.
TRIV: Process TRIVIAL diagnostic.
TRUBL: Abort compilation (test mode)
WARN: Process WARNING diagnostic.
WCODE: Write code to prebinary.
=XLIB: Define external (library) routine name.

j. Symbol Definitions

Tables of symbolic values pertinent to compilation.

PASS=: Table manager phase information
IOCAD: I/O control code values
ODEF: Token value definitions
CH.: Charmap define/describe.
Micros: ODEF related micros
DUC=: QCG EMIT constants
PSUD: Pseudo instruction values
IPSUD: Pseudo label instruction values.

k. Table Definitions

DESCRIBE/DEFINE for data structures of FTN5.

PB.: Prebinary table. Comdeck COMSPBD.
WA.,
WB.,
WC.: Symbol table. Comdeck COMSSYM.
CA.,
CB.: Common block name table. Comdeck COMSSYM.
FP.: Formal parameter table. Comdeck COMSSYM.
TB.: Token buffer.
DH.,
D1.,
D2.: Dimension table.
DO.,
DP.,
LA.,
LC.: Block structure table.
EC: Equivalence class table.
EQ.: Equivalence item table.

```
TH.,
SP.,
TP.:    Turple descriptions (IL).
AG.:    Assign statement label table.
NG.:    Namelist table.
IA.,
OA.:    Actual parameter list table.
DA.:    Data statement table.
DI.:    Data constant table.
II.:    Input item list table.
XR.:    Cross reference table.
CR.:    Cross reference attribute letters.
EP.:    Entry point table.
EH.,
EF.:    Entry parameter list table.
SB.,
SR.,
IS.,
LB.:    Block tables.
F.PIK:  Machine opcode description table.
MO.:    MOD cell description.
PM=:    Parser mode values.
AM.,
AC.,
AS.:    Parser interface cells (ARGMODE, ARGCOMA, ARGMISC)
        definitions.
KW.:    Keyword table.
SF.:    Statement function table.
IT.:    Intrinsic table.
INTF=:  Intrinsic table entry macro.
```

1.2 CMPLTXT: Compiler Products Assembly Text

Abstract: CMPLTXT consists of macros and micros used by the CCG interface decks.

Interface: CMPLTXT is used to assemble the CCG interface decks.

Data Structures

a. Options

See FTN5TXT (1.1). Comdeck OPTIONS.

b. Data Structure Definition Macros

See FTN5TXT (1.1). Comdeck COMADEF.

c. General Compiler Macros Comdeck CCOMGCM.

LXQ:	Left shift redefine OPDEF. Eliminates zero shifts.
RPVDEF:	Entry definition for REPRIEVE.
RPVFWA:	FWA definition for REPRIEVE.
LISTL:	List one line. RM interface.
NUPAGE:	Page eject. RM interface.

d. Instruction Descriptor Fields. Comdeck CCGILFD.

CCG describe/define for the IL. (At BRIDGE time)

e. OP Code Descriptors (Comdeck OPRDEFS.

CCG describe/define for the IL. (Return from CCG)

f. SLIST Instruction Descriptors Comdeck PSODEFS.

Describe/define for CCG IL pseudo opcodes.

NOTE: OPRDEFS and PSODEFS calls are imbedded in CCGILFD.

g. CCG Interface Macros

WRITEP: Write pseudo opcode to SLIST file.
ADDWRD: Add a word to a managed table.
ALLOC: Allocate managed table space.

1.3 CCGTEXT: Common Code Generator Assembly Text

Abstract: CCGTEXT consists of macros, micros and assembly constants used by the CCG decks.

Interface: CCGTEXT is used to assemble the CCG decks.

Data Structures

a. Options

See FTN5TXT (1.1). Comdeck OPTIONS.

b. Data Structure Definition Macros

See FTN5TXT (1.1). Comdeck COMADEF.

c. RM I/O

See FTN5TXT (1.1). Comdeck FA=DEFS.

d. General Compiler Macros

See CMPLTXT (1.2). Comdeck CCOMGCM.

e. General Macros (Less so than CCOMGCM)

LEN:	Count number of names in micro list.
MXX+X:	OPDEF for maximum function.
MXX-X:	OPDEF for minimum function.
BIT:	Set symbol to power of 2.
CALL:	Issue RJ to external subroutine.
ENTRY.:	Define entry point and contents.
EQENT:	Generate an entry point and EQU.
MOVE:	Move a block of data.
PLUG:	Modify compiler code during execution.
NAME:	Define local subroutine entry point.
SETCORE:	Set block of memory to a given value.
SETZERO:	Set block of memory to zero.

f. CCG Debug Macros Comdeck CCGDBGM.

PRINT:	Print the contents of a list of locations.
TRACER:	Define routines/phases to trace.
TRACE:	Conditionally snap table contents.
SNAPRL:	Interpretive dump of IL.
DCALL:	Call debugging routine.
REGSNAP:	Snap registers at entry points.
SNAPT:	Snap table using pointers.

g. Interactive Debug Macros Comdeck DBG=MAC.

G7

BREAK: Place a breakpoint.
FRK=: Generate frequency parameter list.
REG: Register snapshot.
RGR=: Generate register parameter list.
SNAP: General snapshot interface.
SNG=: Generate indirect address fields.
USF=: Generate user flag parameter cell.

h. Instruction Descriptor Fields

See CMPLTXT (1.2). Comdecks CCGILFD, OPRDEFS and PSODEFS.

i. Intermediate Language Descriptors

BI.: BIT table
ML.: MOD list
T.: Temporary equivalence table
LC.: Label change table.

j. Host Processor Interface

A series of EQU's defining external cells in the host processor (FTN5).

k. Comdeck COMSSYM

Symbol and other table descriptions. Described in FTN5TXT (1.1).

l. CCG Macros

CCG.SST: Generate SST call.
SETB1: Set B1=1.
ADDWRD: Add a word to a managed table.
ALLOC: Allocate managed table space.
PROCESS: Define processor addresses.
EXT=: Declare names of externals.
TABLES: Declare names of dynamic tables.

2.0 CRADLE ROUTINES

The decks grouped here are those that are common to the front end and rear end. Some are used by the CCG overlay, some not, but the multiple use in several overlays determined the location within the routine descriptions.

Several decks perform similar functions, although utilized by only one overlay. They can be thought of as cradle functions and are grouped here for convenience. These include initialization routines, snap routines (including IDP, test mode only), and linkage routines.

Usage of the routines/decks in question is described in the pertinent interface section.

2.1 FTN: Global Cells and System Interface

Abstract: FTN contains compiler installation parameters, intermixed COMPASS communication cells, file management tables, compiler global cells, control statement option cells, system interface subroutines, compiler overlay loader and termination routines.

Interfaces: FTN consists of static and dynamic information, some of which is bound at installation time (system defaults, installation defaults), some at control statement recognition time, and some by a combination (defaults for missing control statement parameters). Initialization is performed on a one-time basis by INITOO and the information survives all future overlay loading (including COMPASS). The system interface routines are environment dependent and are bound at installation time. FTN is present in all overlays.

Data Structures

Most of FTN's data structures are simply cells, which contain flags, file names, etc., which were directed by the control statement (or defaulted). In sequential order, these are:

a. COMFCIP. The Comdeck which contains micros and equs for installation defaults. Provides default settings for control statement parameters and for CIO interface buffer lengths.

b. COMPASS Interface. Provided by Comdeck COMPCOM plus micros, equs and entry declarations to provide other decks an interface to COMPCOM's cells. This area contains cells of information used by both FTN5 and COMPASS (when intermixed).

c. File Management Tables. FET/FIT macro expansions for standard files utilized by FTN5. These include INPUT, OUTPUT, ERROR, LGO, Prebinary, Intermediate File and Cross Reference File.

d. Control Statement Flags. Calls preset with information from COMFCIP and, dependent upon the control statement, reset and/or reformatted by INITOO.

e. Title Line Templates. Used for source listing. Appears here because the title line information is needed for error output regardless of the requirement for source listing.

Routine Descriptions

a. LDCOM. Forms loader request for loading COMPASS (1,0) overlay.

b. LOVER. Forms loader requests for loading the compiler overlays.

   Determines which overlay is to be loaded, and from what source (file or library). Sets up the request registers and exits to LOV.

c. LOV. Load overlay. Accepts information from LDCOM or LOVER and sets appropriate communication area cells/flags. Clears REPRIEVE and SPY requests as necessary and performs the overlay load request. After control is returned, SPY is cranked up as necessary, and control is transferred to the requested overlay. Loader failure results in an abort.

d. STOP. Return point from intermixed COMPASS. Performs some restoration and falls through to ...

e. LDPRI. Load Primary overlay. This routine reloads the FTN5 overlay which COMPASS displaced. Also used by INITOO to fetch the (1,0) overlay or (2,0) when OPT>0.

f. MEMERR. Outputs insufficient FL message.

g. IDPCHK. Breakpoint check for each overlay loaded. Determines if break requested for the current overlay and, if so, transfers to IDP for processing. Test mode only.

h. ONSPY. Interface to PPU program SPY. Turns SPY on, as necessary.

i. OFFSPY. Ditto, turns SPY off. Both routines conditional upon .SPY,ON.

j. COD. Convert octal to DPC. Comdeck COMCCOD. Converts an octal constant of up to 10 digits into display code. Leading zeros are suppressed and both right- and left-justified results are available. Space filled.

2.2 UTILITY: Common Utility Routines

Abstract: UTILITY contains routines of a general, service nature.

Interfaces: All routines in UTILITY are provided via the common comdecks and reside on a secondary PL. The deck provides entry names for the routines. Most entry names are '=' suffixed. Present in all overlays.

Data Structures

None of note.

Routine Descriptions

a. CDD Constant to decimal DPC conversion. Converts a binary constant (up to 10 digits) to space filled DPC. Left- and right-justified forms returned. Comdeck COMCCDD.

b. DXB. DPC to binary conversion. Converts DPC representation of an octal or decimal constant to binary. Constant (DPC) may be 8 or 9 digits, depending on whether it is suffixed with base indication (B or D). Comdeck COMCDXB.

c. FA=SEI. Set file environment table for CIO directed I/O, or record manager I/O (conditional assembly). The CIO flavor sets the buffer address in the FET. The RM version initializes holding buffer addresses in FIT and pseudo FET. Comdeck FA=SET.

d. MVE. Move block of data. Given a source address, word count and destination, a block of data is relocated. Move may be either direction, is performed word at a time and requires two styles of move in order to preserve the moved data. (Style dependent upon upward or downward move.) Comdeck COMCMVE.

e. RPV. Reprieve Processor. When the program (FTN5) is abnormally terminated, RPV takes control to issue dayfile messages concerning nature of error, location of error (and in test mode, the DECK (IDENT) in which the error occurred). Output mode files are finished and the error condition is reset to provide EXIT condition processing. Comdeck CCOMRPV.

f. ERA. Find Relative Address. Used by RPV to compute relative address within an IDENT (DECK, ROUTINE) when a table of addresses is provided. Resides on comdeck CCOMRPV.

g. MNS. Move Nonoverlapping String. This routine will move an arbitrary bit string from one location to another in core. The source and destination locations may start at any location within a word and may cross word boundaries. Source area is unchanged, and the destination area may not overlap the source in any manner. Comdeck COMCMNS.

h. SBM. Set block of memory to a given value. The value is set into the first word of the block and the set continues word at time for the length of the block. Comdeck COMCSBM.

i. SEN. Converts trailing '00' characters to '55' blank characters. Based on a Mansfield algorithm. Comdeck COMCSFN.

j. SSI. Shell Sort Table. Sorts a table of one-word entries into ascending order. Standard shell sort algorithm. Comdeck COMCSST.

k. WOD. Convert binary word to DPC. This routine converts binary (octal) data to DPC. Algorithm by C.R. Willis. 64-character set. Comdeck COMCWOD.

l. ZIB. Converts all zeros (binary) to blanks. Algorithm from Mansfield. Comdeck COMCZTB.

m. CIO. I/O function processor. Provides the interface to system PP I/O processor routines. Comdeck COMCCIO.

n. RDC. Read coded line. Reads a zero byte terminated coded line from a buffer. Comdeck COMCRDC.

o. RDW. Read words to working area. Transfers a given number of words from a CIO buffer to a user-defined working area. Comdeck COMCRDW.

p. RDX. Read exit. Exit routine from RDW to user. On comdeck COMCRDW.

q. LCB. Load circular buffer. For RDW. On comdeck COMCRDW.

r. WTH. Write coded (Hollerith) line. Transfers coded line from working storage to CIO buffer. Comdeck COMCWTH.

s. WTO. Write one word. Writes one word to a file (actually to a buffer, which is output as necessary). Comdeck COMCWTO.

t. WTW. Write from working buffer. Transfers data from a working buffer to a CIO buffer. Comdeck COMCWTW.

u. WTX. Write exit. Exit routine from WTW to user. On comdeck COMCWTW.

v. DCB. Dump Circular Buffer. For WTW. On comdeck COMCWTW.

w. FA=CLO. Close a file, if open. RM interface. Comdeck FA=CLO.

x. FA=EOF. Write end-of-file. RM interface. Comdeck FA=EOF.

y. FA=EOR. Write end-of-record. RM interface. Comdeck FA=EOR.

z. FA=FLSH. Flush file holding buffer. RM interface. Comdeck FA=FLSH.

aa. FA=OPE. Open a file. Prevents (dishonors) redundant open attempts. RM interface. Comdeck FA=OPE.

bb. FA=RDC. Read coded line. Transfers coded line to working buffer. RM interface. Comdeck FA=RDC.

cc. FA=RDW. Read words to working buffer. RM interface. Comdeck FA=RDW.

dd. FA=RWX. Rewind file. Suppressed for scratch files. RM interface. Comdeck FA=RWX.

ee. FA=WTH. Write coded line. To a file. RM interface. Comdeck FA=WTH.

ff. FA=WTW. Write words. From working buffer to sequential file. RM interface. Comdeck FA=WTW.

NOTE: Only the CIO or RM I/O routines are assembled, depending on the system.

2.3 PUC: Program Unit Controller and Support

Abstract: PUC contains data and routines of a general nature, which are required by all (or most) overlays. Performs program unit initializations.

Interfaces: Contained in all overlays.

Data Structures

Data structures of PUC are mostly communication cells, used for information which has validity for the duration of a program unit.

a. Tables. A series of cells (four tables), used by the table manager (and any routine which accesses the managed tables). Provides entries T.xxx (table FWA), T=xxx (table size) and associated tables regarding the growth rate of individual tables and (test mode) table names for the IDP interface. Comdeck COMSTAD is used to provide table information for CCG shared tables.

b. Local Block Table. Local block ordinal table. Used by code generation, assembler, etc. Provided by comdeck COMSLBT.

c. Common Cells. These are flags and indicators which reflect the current state of the compilation. The cells here are typically things which must survive secondary overlays (OPT>0) and that pertain to front end/code generation/assembly. There are also working copies of control statement option cells which may be altered by C$ directives.

d. Short Constants. A table of commonly used short constants (e.g., 0, .true.) in TP. format. Used by front end processors mostly, but some have CCG interface application.

e. E.SORD. A table of symbol table ordinals for symbol table entries made by the compiler as part of the initialization process.

f. ERRTYP. A table of diagnostic message labels used by front and rear end diagnostic production.

g. E.BIK. Instruction description vector. A table of COMPASS CPU instruction templates, used in formatting object listing and determining instruction form.

Routine Descriptions

a. PUC. Program Unit Controller. PUC is entered after overlay initialization and provides some startup processes and directs the flow of compilation. Functions include: setting up the listing pagination for the current program unit, clearing the compiler scratch files, determining if the program unit is a FORTRAN or COMPASS coded routine (if COMPASS, transfer to LDCOM for COMPASS overlay load is made) and successive calls are made to the front end, code generator and rear end loaders. (Whether actual loading is performed is a function of the OPT level.) Upon completion of compilation, PUC computes some program unit statistics and outputs such things as memory used and error totals.

b. ENDFTN. Terminate compilation. Performs final housekeeping chores upon completion of compilation, outputting dayfile messages, turning off SPY, etc. Contains abnormal termination code and code to initiate binary execution when the GO option is present.

c. CPTIM. Computes elapsed CPU time (via call to TIMER) and converts result to DPC.

d. TIMER. The routine which interfaces the system clock and converts elapsed time into milliseconds.

e. WFA. Wait File Actions. The CIO interface wait routine.

f. CFD. Convert an integer (30 bit) to display floating. Comdeck COMCCFD.

g. CAF. Close All Files. Called by PUC upon compilation finished. Performs cleanup function of closing opened files and eviction of compiler scratch file.

h. COF. Close output file. Called by CAF to reset print density (if necessary) on sequential output files.

i. ECB. Evaluate constant character bias. Using symbol table information, ECB calculcates the actual bias for constant character array subscription. Comdeck COMFECB.

j. ECS. Evaluate constant substring. ECS calculates bias and substring length for constant character substring references. Comdeck COMFECB.

k. GCL. Get character length. GCL extracts the associated length attribute for type character symbols (or if assumed length, the VD. tag for the defining length).

l. GMC. Get More Core. Provides system interface for MEM requests when current FL is insufficient for managed tables. If the maximum FL has not yet been attained, a provided (user) increment is requested from the system.

m. LJS. Left-justify statement label. This routine is used by diagnostic production and rear end listing routines to adjust and space fill statement label DPC representation. LJS is necessary because of the symbol table format for label names (right justified).

n. MTD. Move Tables Down. MTD compresses the contents of the managed table area into the low core portion of the managed table area. Used by the FTN5 (not CCG) table manager for reallocation and by CCG.

o. MTU. Move Tables Up. MTU compresses the contents of the managed table area into the high core portion of the managed table area. Used by CCG.

p. PIA. Process instruction address. PIA converts a binary number to octal DPC, with leading zero suppression and a 'B' suffix. Used for object listing address fields and wherever this type of conversion is needed.

q. PES. Print error summary. Loops through error count information, producing listing of error counts by error level. Output to console and dayfile.

r. PCS. Print Core Statistics. Output the current FL used by the compiler. Called when a MEM request is made.

s. WHL. Write Header Lines. WHL is called by WOF when a new page of listable output occurs. Performs some conversion and template plugging prior to outputting the header.

t. WOF. Write output file. The standard interface to the coded output routines (UTILITY). WOF determines the nature of the request, precedes the output with blank lines as required, determines if page size is exceeded (and increments pagenation information and calls WHL) and passes the output information to the relevant system interface routine for output.

2.4 Linkage Decks

H2

FTN5 utilizes common front and rear ends which send input to and receive output from two dissimilar code generations: QCG and CCG.

The differing overlay structures (Sect. A, 1.0) force different control flow methods for the two code generators. Typically QCG is loaded as one overlay and remains in core for the duration of compilation while CCG loads successive secondary overlays for phase transition. The two code generators require some differing information and processing. This can involve different processing routines to perform the 'same' function.

In QCG mode, the presence or absence of the map and object listing routines change requirements for routine calls.

To keep the front and rear ends truly common, the system of link decks was devised. Where functions are needed by one but not both code generation modes, the function is placed in both link decks. Where the function is required, code to perform the function is provided. Where the function is not required, a stub is placed, providing a 'do nothing' subroutine. This allows the common portions of the compiler to just request the function, regardless of code generation mode, and also allows the overlay which didn't require the function a core savings.

The linkage routines vary from overlay to overlay, and technically are not really cradle routines. But since at least one link routine is present in every overlay structure, the routines logically belong to the cradle set.

2.4.1 QCGLINK: QCG Mode Linkage

Abstract: QCGLINK provides stubs and routines for the QCG code generation mode.

Interfaces: QCGLINK is part of the (0,0) and (1,0) overlay.

Data Structures: None

Routine Descriptions

a. FEL: Front End Loader. No loading actually performed. Transfer is made to FEC and an entry point is provided for return. Provided for compatibility with CCG which requires a load of front end routines.

b. CGE: Check code generator errors. Stub only.

c. DER: Detect extended range do loops. Stub only.

d. LPE: Link possible entry do loops. Stub only.

e. MDD: Mark do parameters defined. Stub only.

f. PDC: Process divide by constant. Stub only.

g. BCT: Blanch constant table. Stub only.

h. MAL: Mark loops entered. Stub only.

i. PCA: Process CAC table. Stub only.

j. PAT: Preprocess A-P list table. Stub only.

k. CGL: Code Generator Loader. Stub only. In QCG mode generation, code generation is performed concurrently with front end processing,; thus, no need for a transfer here.

l. REL: Rear End Loader. Again, no load, just transfer. If the prebinary file is still in core, it is flushed to disk for CCG compatibility. An entry point is provided for return.

m. PDI: Publish data to IL file. The QCG version copies DATA statement output from T.DATS to the prebinary file (table or file).

n. PIS: Publish IL Segment. The QCG version calls CAI to compile instructions to the prebinary file (table or file).

2.4.2 CCGLINK: CCG Mode Linkage

H4

Abstract: CCGLINK provides routines specific to the CCG mode overlay structure.

Interface: CCGLINK resides on the (2,0) overlay.

Data Structures

A table of CCG linkage values is provided.

Routine Descriptions

a. FEL: Front End Loader. Calls the overlay loader processor to load the (2,1) overlay. Provides an entry point for return from front end processing.

b. CGL: Code Generator Loader. CGL prepares the managed table area for use by CCG. If the cross reference file is still in core, it is flushed to disk. The overlay loader processor is called to load the (2,2) overlay. An entry point is provided for return. Upon return, managed table pointers are updated for rear end processing. If syntax errors occurred during front end processing, loading of the (2,2) overlay is suppressed.

c. REL: Rear End Loader. The prebinary table is flushed to disk and the overlay loader processor is called to load the (2,3) overlay. An entry point is provided for return.

d. MAT: Move all tables. Called by CGL to reformat the managed table area for CCG. MAT preserves tables which must survive front end to rear end. Common tables front end - code generator are relocated. All other tables are trashed.

e. UTP: Update Table Pointers. Reverses the transformation of tables common to front end - code generation performed by MAT. Tables are now in a usable form for rear end processing.

2.4.3 ZEROLNK: 0,0 Overlay Linkage

Abstract: ZEROLNK provides stubs for routines pertaining to map and object listing, which are not present in the (0,0) overlay.

Interfaces: ZEROLNK resides on the (0,0) overlay. Its purpose is to allow a common rear end regardless of output listing parameters.

Data Structures: None

Routine Descriptions

The following routines are all stubs.

a. PIK: Print instruction conversion.

b. MAP: Map listing.

c. LUS: List Unit Statistics.

d. EIN, EIN.MAP, EIN.OL: Entry points to provide pseudos for the LWA of the overlay.

2.4.4 RLINK: Rear End Linkage.

H6

Abstract: RLINK provides routines, stubs and data structures for the rear end processor in a CCG context.

Interfaces: RLINK resides on the (2,3) overlay.

Data Structures

The data structures described below are used to transform CCG output into a format understood by the rear end processor.

a. RTT: Register Translation Table. Conversion matrix for CCG coded register designations to actual register numbers. Used by CII.

b. HTT: H field translation table. Used by CII.

c. OPRDEFS: CCG IL instruction definitions. A table used for converting CCG IL into machine instruction (prebinary) format by CII. Comdeck OPRDEFS.

Routine Descriptions

a. LDB: List Deferred Buffer. A stub, which in test mode provides a blowup if action was to have been taken.

b. CGE: Check code generation errors. The code generators have no diagnostic capability in FTN5. CCG, however, does diagnose some error conditions. This interface uses the rear end diagnostic capability to output messages required by CCG.

c. CII: Convert Issued Instructions. This routine transforms the CCG intermediate file code into prebinary format required by the rear end processor. Processing is performed for one instruction/code at a time.

d. BCI: Convert constant table. The CCG constant table is converted from bias arrangement into the actual constants (T.CUT to T.CON transformation). Unused constants are eliminated.

e. CFP: Check Formal Parameters. Used by CII to process formal parameters in the CCG IL - prebinary transformation.

f. SMB: Set materialize bit. Stub only. H7

g. PAI: Process AP-list tables. A transformation of CCG. AP-list constants into prebinary form required by the rear end processor.

h. PCA: Process character address reference constants. Transforms T.CAC from the CCG constant style to the prebinary form.

i. SCS: Scan table with supplied mask. This routine is used by front and rear ends, not by code generator. Description in deck FEC. On comdeck COMFSCS.

2.4.5 LISTLNK: Listing Linkage

H8

Abstract: LISTLNK provides the program unit statistic output routine. Exists to save space in the (0,0) overlay.

Interfaces: LISTLNK appears where program listings are required and resides on the (1,0) and (2,0) overlays.

Data Structure: Program Unit Statistic template.

Routine Description:

LUS. List Unit Statistics. LUS is the subroutine called by PUC to provide the calculation and formatting of final program unit statistics. Functions include converting binary values to listable DPC, formatting variable data into the statistic template and calculation of some values.

2.5 PEM: Print Error Messages

Abstract: PEM accepts a diagnostic request, and if output is required, formats the text and outputs the diagnostic.

Interfaces: Diagnostic processing is provided only for the front end and rear end. Thus PEM is contained in the (0,0), (1,0), (2,1) and (2,3) overlays. Although not a 'pure' cradle routine, PEM is considered universal enough to be included with those routines.

Data Structures

The following data structures are used by PEM. All are described elsewhere.

a. Assembly Constant Table: Described in FERRS.

b. ERRSKEL: Diagnostic skeleton definition, described in FERRS.

c. ERRTYP: Described in PUC.

d. The actual error texts are described in FERRS.

e. CHARMAP: Described in FEC.

Routine Descriptions

a. ANSI: ANSI diagnostic blockage. This routine is a filter used to stop processing of ANSI diagnostics when they are not required. Switch is set based on control statement parameters.

b. MDERR: Machine dependent diagnostic blockage. Works exactly like ANSI.

c. PEM: This routine has three flavors, depending on whether variable text information is to be processed. PEMS is an entry used to process cells FILL., FILL.2, FILL.3. All three cells are assumed to be preset in L format. PEMS determines the length of the information string in each cell, and appends the applicable length character. PEMV assumes FILL. is to be set with the current contents of the token buffer, up to 10 characters. A transformation of tokens into characters is made, using CHARMAP, until 10 characters are obtained, or the end of the token buffer is encountered.

When PEM proper is entered, all variable information has been formatted. The error level of the current diagnostic is determined, and a count of diagnostics at that level is incremented. The control parameter error level cell is checked to see if this diagnostic need be printed. If so, the proper prefix label is selected and moved to the buffer area. Each diagnostic word is fetched in turn, its length determined, and the proper number of characters are merged into the buffer. In some cases, page width will require splitting the diagnostic into two lines. PEM handles this. Upon completion of the buffered build, the diagnostic is output to OUTPUT/ERRS file(s).

d. PDM: Print diagnostic messages. FTN5 has two types of diagnostic return. One method is a provided return address. The second is an implied return. PDM provides the mechanism for the implied return. PDM fetches the PEM entry relevant for the diagnostic and jumps to that return. The diagnostic return is set for PDM, which in turn returns to the caller.

e. UEC: Update error count. Used by PEM to calculate and update error counts. Also processes possible error termination cases.

f. SVR/RSR: The decision was made to save all registers when processing diagnostics. The overhead was felt justified by the freedown allowed the routines using the diagnostic processor. SVR= and RSR= save and restore all registers. Comdeck COMQSVR.

2.6 ALLOC: Table Allocation (Managed Tables)

Abstract: ALLOC is the managed table manager for the front end and rear end.

Interfaces: ALLOC appears in overlays (0,0), (1,0), (2,1) and (2,3). It is not a 'pure' cradle routine, but is fairly universal.

Data Structures

ALLOC operates on two data structures: table pointers and tables.

a. Table Pointers: The table pointers consist of two tables of cells, T.xxx which gives the FWA of the table xxx, and T=xxx which contains the length of xxx. A pair of these cells exists for each managed table.

b. Tables: The managed tables are in high core. The starting location varies with various overlay structures. In general, the core used for the initialization decks (INITyy) is available for managed tables. The end of FL delimits the table area, although this limit may be extended by MEM requests. The managed table area and relationship to the pointers is depicted below (Fig. 2.1).

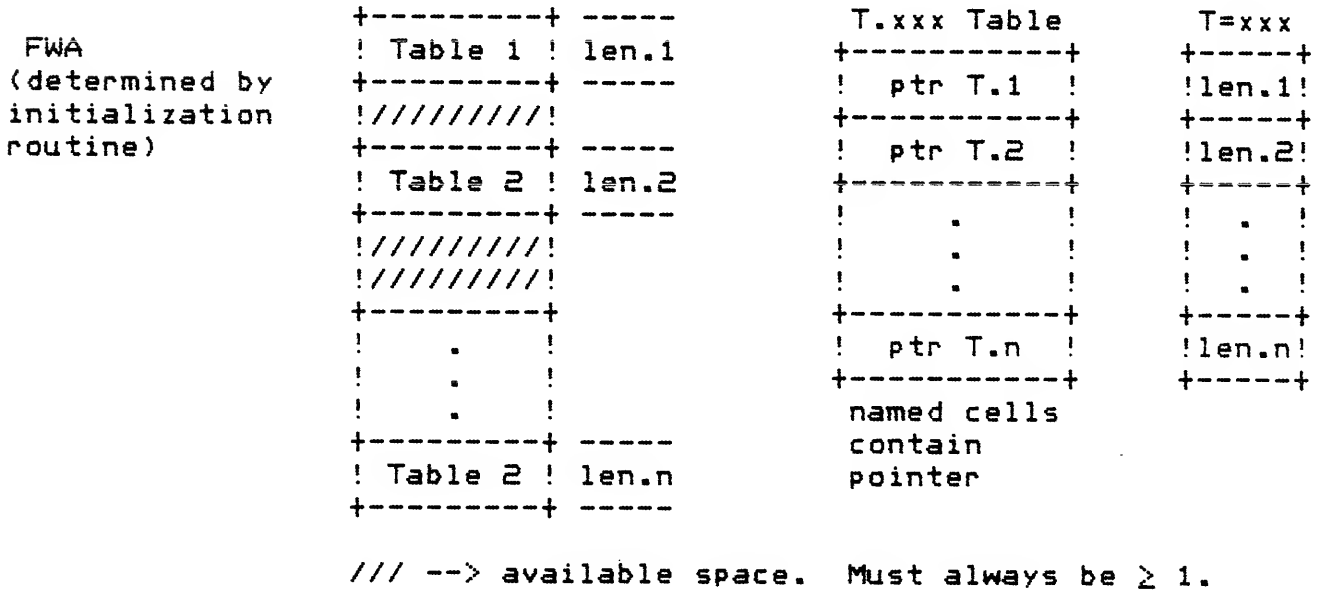


Figure 2.1

a. ADW: Add word. Calls ALC to obtain a word for the required table. Stores the datum into the space made available.

b. ALC: Table manager and allocator. This routine should always be called to add any data to any managed table. Disasters can occur if this rule is not observed.

ALC is passed the table to allocate and the number of words to extend the table. If there is sufficient available space following the table being allocated, the length of the table is updated and ALC is through. (Check listings for return conventions.)

When space is not available, a table crash has occurred. The managed tables are compressed and then reexpanded, reallocating the available space. The amount of free space left between tables during the reallocation is dependent upon the table in question and the current processing state of the compiler. At various points in front end and rear end processing, registers are used as table pointers. ALC supports this madness by allowing up to one lock register. If a lock register is used, the register contents is converted to an offset into the associated table. Upon completion of allocation, the offset is converted back to an address.

If sufficient room in the managed table is not available, GMR is called to try to obtain the room. It will do so, or abort the compilation. Concurrent with moving the tables around, the T.xxx table is updated to reflect the current state of the individual table FWAs. Upon completion of the reallocation, the T.xxx reflects the state of the managed table area, and the specific T=xxx has been updated with the additional allocated space.

c. GMR: Get More Room. GMR is called by ALC when there isn't sufficient managed table space to satisfy the current request and to allow space for expected allocations in the near future. GMR first tries to obtain the necessary space by flushing some tables to disk files. If that is not sufficient (or has already been done), a call is made to GMC to make a MEM request. If the MEM request is disallowed, but the current allocation was satisfied, GMR will return and attempt to continue compilation. If the specific request is not satisfied, compilation is aborted.

d. PIA: Print Threshold Alarm. Called by GMR when a specific allocation was barely able to be satisfied, no more MEM are allowed. Test mode only.

e. PIS: Print table statistics. Test mode debug device to track table crashes. When table crash occurs, the table being allocated and status of the managed table area are output. Selected by the SNAP=T control statement option.

2.7 Snap Interface Routines

H14

The snap routines provide interface to IDP (interactive debug processor) and format table dumps and snap output. There is a separate routine for the front end, CCG and rear end processors to provide the formatting needs of these processors. The snap routines are test mode only. These decks/routines are not true cradle, but one of the routines appears in every test mode overlay.

2.7.1 FSNAP - Front End Snap Package

Abstract: FSNAP provides IDP interface and dump formatting for the front end processor and for single overlay processors.

Interfaces: FSNAP interfaces heavily with IDP, and the break mechanism therein. It appears as part of the (0,0), (1,0) and (2,1) overlays, test mode only.

Data Structures

a. COMSIDP: IDP interface macros, micros and definitions. Described in the IDP deck (2.7).

b. COMSQCG: Quick code mode structure definitions. Used for formatting dump information. Described in QCGC deck (4.1).

c. COMSIDK: Data structures and routines pertaining to token generation. Used in formatting snaps of the token buffer. Described in LEX deck (3.3).

d. Other: Other data structures are described with the routines that utilize them.

Routine Descriptions

a. CID=: Convert token type to DPC. This routine takes the token type passed it, and returns the corresponding DPC of the token. Performed via table lookup in CHARCMAP (described in deck FEC (3.1)).

b. ETH: Format table heading. Performs table lookup for table names, origins and sizes. (NAMES, BASES, SIZES in PUC (2.3)). The binary numeric data is converted to octal DPC. Used when dumping a table.

c. LIB: List token buffer. An interface routine for LTK (list tokens). Saves and restores registers and informs LTK of the nature of the request.

d. SN.EMT: Snap emitted turples. Formats output for a single output turple (front end format). Call the formatters for the turple header and turple operand (twice).

e. SN.PAR: Snap parse file. Similar to SN.EMT, except that the entire contents of table T.PAR are formatted. Action provided by SN.EMT is performed in a loop.

f. SN.PSYM: Snap parse file symbol. This is the formatter for the operand fields of a turple. Breaks the TP. format into component parts and formats for output the DPC (if symbol table entry) or octal DPC (intermediate or short constant) as well as mode and other information.

g. SN.POP: Snap parse file operator: The formatter for the turple header (operator) word. Breaks the TH. format into component parts and formats DPC of the operator, turple mode and the status of the operands. Both SN.POP and SN.PSYM utilize output templates into which the DPC format information is plugged. There are also small tables of DPC values for mode and operand information.

h. DAT: Dump a table. Interface to IDP core dump routine (DCM=). Formats table length, name and origin for header. Determines if symbol table is to be dumped and selects either DCM= or DSY for the table dump. Comdeck COMDDMT.

i. DMT=: Dump tables. The driver for DAT. Determines how many tables are to be dumped, and successively calls DAT for each table. Comdeck COMDDMT.

j. DSY: Dump symbol table. Formatter for dump of symbol table. Breaks up the symbol table WA., WB. and WC. words into segments as defined by those formats. Provides a DPC and octal DPC mixture of formatted information. Comdeck COMDDMT.

k. COMDTOK: This comdeck contains routines and structures used in producing tokens from source. FSNAP uses the routine to disassemble tokens to provide snap of token buffer. The description for COMDTOK routines and data structures is in LEX (3.3).

2.7.2 CSNAP: CCG Bridge Snap Package

Abstract: CSNAP provide IDP interface and dump formatting for the CCG/Bridge environment.

Interfaces: CSNAP interfaces IDP and the break mechanism. Resides on the (2,2) overlay, test mode only.

Data Structures

CSNAP utilizes COMSIDP, described in IDP (2.7).

Routine Descriptions

CSNAP utilizes the following COMDDMT routines, described in FSNAP (2.6.1).

a. DAT
b. DMT=
c. DSY

2.7.3 RSNAP: Rear End Snap Package

12

Abstract: RSNAP provides IDP interface and dump formatting for the rear end processor in the CCG environment.

Interfaces: RSNAP interfaces IDP and the break mechanism. Resides on the (2,3) overlay, test mode only.

Data Structures

RSNAP utilizes COMSIDP, described in IDP (2.7).

Routine Descriptions

RSNAP utilizes the following COMDDMT routines, described in FSNAP (2.6.1).

a. DAT
b. DMT=
c. DSY

2.8 IDP: Interactive Debug Package

Abstract: IDP consists of common routines/structures used by all products/programs which utilize IDP and FTN5 specific code for compiler interface to IDP. IDP users are referred to the IDP reference manual.

Interface: IDP (the deck) interfaces IDP (the program). IDP usage is beyond the scope of this document. IDP resides in the (0,0), (1,0) and (2,0) overlays, test mode only.

Data Structures

a. COMAIDK: Token generation language macros. Described in LEX (3.3).

b. COMSIDP: This IDP provided comdeck contains macros, micros and structure definitions used by IDP and interfacing routines. It serves the function of a text, but is assembled as part of routines which require it. Macros include those for generating keyword (IDP) tables, entry tables, formatting routine parameter set up. IDP symbols are defined here.

c. COMSIDK: Data structures and routines pertaining to token generation. Described in LEX (3.3).

d. Other: Other structures will be described with their pertinent routines.

Routine Descriptions

a. SI=ABI: Abort IDP. Restore registers and evoke system abort.

b. DXP: Dump exchange package. Compiler specific. Provides a NOS/BE like dump for SCOPE 2. Needed because SCOPE 2 has no reset, and the register dump is the status of the register after reprieve.

c. PIO: Print table origins. Compiler specific. Formats managed table information for output. Outputs the information, including a list of pseudos applying to each table.

d. IEX: Transfer exchange package registers. The contents of the saved exchange package (registers) are moved to the COMCSVR save area. Restore of registers is then from the saved exchange package. Compiler specific.

e. UIO: User IDP owncode. Compiler specific. Interfaces IDP break request. If a new overlay has been loaded since the last IDP call, the break point table is cleared. (An IDP call is executed at each overlay load [when in BREAK model for this purpose.)

f. URO: User IDP register owncode. Compiler specific. URO selects or deselects register snaps, based on requested snap option information.

g. USO: User IDP snap owncode. Compiler specific. USO selects or deselects core snaps, based on requested snap option information.

h. USY: User IDP symbol search. Compiler specific. USY provides access to an IDP symbol search routine. An RJ is constructed to the search routine, which is overlay dependent.

i. LFM: Local file manager. Provides system interface to PPU program for file management. Comdeck COMCLFM.

j. COMCIDP: This comdeck is the interactive debug package proper. Routines and structures described here are not under project control

1) Structures

ADR=RJ: Return address of calling routine.

APL: IDP parameter list, from the IDP call.

BC=BRAD: Previous contents of break address.

E.BDO: FET and buffer for batch debug output.

E.IDI: Line image FET for interactive debug input.

E.IDO: FET for interactive debug output.

EW.SUR: Register save area.

IDFLAG: Coded flag giving information about current break/snap request.

EW.POT: Parsing operator/operand table. Used by IDP to parse expressions in IDP requests.

EW.PAST: Parsing stack. Used for the shift reduction of operators during parsing of expressions in IDP requests.

EW.RPN: Reverse Polish stack. Used for operands and reduced operators to form reverse Polish notation used during parse of expressions in IDP requests.

SNAPLNE: Output line image buffer.

Cells: IDP provides a large group of internally used storage cells to provide flags, status information and values used by IDP in processing requests.

ICE: User/token generation communications area.

EW.SCI: Statement control token table.

EW.KEY: IDP keyword table. Used to drive processing of request parse. Broken into IDP commands, break commands, step commands, and has associated subkeyword table for step and output options.

EW.ERR: Diagnostic texts for IDP messages and FW.SER system errors.

IDPBA: IDP break address and break contents IDPBC tables. Parallel.

IDPIMP: User temporary table.

IDPSET: IDP set value table.

IDPXFT: IDP most recent transfer address table.

2) Routines

IFR: IDP freeze recovery/restart. This really isn't a routine. It is copied to the file F.FRZ as the first record when restart is required from a freeze requested by the host program. The routine is executed by transferring control to the F.FRZ file.

SYS: System request interface routine. Processes system calls.

RHN: Read host into hole. Reads a frozen host from F.FRZ into the hole created by IFR. IDP processing continues.

IDP: The interactive debug routine. This routine controls the processing of IDP requests. When IDP is entered for the first time, initialization is performed. Afterwards, this initialization is bypassed. The IDP user requests are then processed as they are encountered and the proper processing routines are called. After completion of the request, control is returned to the user.

REG: Register snapshot. REG provides an octal DPC dump of all, or selected, register contents.

SNP: Core and register snapshot. SNP provides an octal DPC dump of selected core locations and requested registers.

SCH: Selection control routines. Provides small utility routines which process brief mode toggle switch, output diagnostics, process ABS directives, process break directives, process code directives, connect files, process DPC directives, disconnect files, and processing of requests, freeze processing, jump address processing, option processing, output processing, register dump processing, REL directive processing, SET directive processing, SNAP directive processing, status processing, STO directive processing, step mode processing, unbreak processing, unset processing, WHERE directive processing, XFER directive processing and STRN directive processing. All these routines are called by IDP when the directive indicated is encountered.

ADZ: Add word to IDP table. ADZ searches the indicated table for the required entry. If not found, the new entry is made and the end of table indicator is reset.

BRK: Break processor. The routine is called when a break point is encountered during execution. Interfaces to IDP for directive request processing.

CAD: Convert address from binary to octal DPC. Used by several IDP routines.

CBC: Check Break Condition. When BRK is entered, CBC is called to check if the break condition was met. If so, break processing commences; otherwise, BRK will revert to caller.

CHK: Check memory address. Determines if the indicated address is accessible by the user.

CIB: Convert integer to binary. Converts a string of DPC digits (decimal or octal) to binary representation.

CLZ: Clear IDP table. The indicated IDP table is cleared to all space available indication.

CON: Connect/disconnect file. System dependent interface to connect or disconnect IDP files. A flavor is provided for SCOPE 2, NOS and NOS/BE.

CSI: Classify statement. Used by IDP to match user directives with the keyword table. This, in turn, determines which SC= routines are invoked.

CXR: Check executive RJ. CXR determines if the IDP executive request had a parameter list, and returns its address if one is present.

DAB: Dump A or B register. Service routine to format contents of A/B register and the contents of the address pointed to by that A or B register. Called by DAR and DSR.

DAR: Dump All Registers. Produces the formatted register dump for all registers.

DAZ: Disassembler. For CODE directive requests. Treats binary as if it were code, and produces a mnemonic assembly listing for the area requested. Uses the table DAZ=PS for formatting the output.

DCM: Dump Central Memory. Converts binary to octal DPC for the area of memory requested. Used for most table dumps and snapshots.

DOD: Dump central memory, octal and DPC. DOD provides two output representations at central memory: octal DPC and alpha DPC.

DSR: Dump Selected Registers. DSR provides register snapshot, as per DAT, but only for registers requested.

DUX: Dump X Register. DUX dumps the octal DPC contents of a specified X register. Called by DAR and DSR.

EAA: Find Absolute Address. Converts address requests of the form 'deck' + offset into an absolute core location. A user provided deck name/address table entry is used.

EAB: Format A or B register. Used to format the contents of an A or B register. Called by DAB.

ELL: Check FWA, LWA and length parameters. FLL tests the legality of the named parameters on any IDP request where they are present.

EOF: Flush Output File. FOF conditionally flushes an output file, depending on the contents (or lack of contents) of the buffer.

ERA: Find Relative Address. Opposite routine to FAA. Takes an absolute address and converts it to the form 'deck' + offset.

ERK: Check frequency parameters. FRK keeps track of the number of times a particular request is honored and compares with the increment value. If at an increment, the snap is honored, else control is returned to caller.

ERZ: Freeze interactive session. Initiates the freeze which IFR eventually restores.

GIL: Generate Indirect Load. Processes indirect loading to the desired level of indirection.

HDR: Print snap header. Outputs the formatted header line indicated.

IEX: Initialize executive. Called when processing an IDP request. Provides initialization required for all requests.

IIE: Initialize Interactive Files. Connects/opens IDP files.

ISI: Initialize Set Table. Initializes IDPSET at start of overlay session.

LBI: List Break Table. Provides a DPC octal listing of current break points set. Form is 'deck' + offset, opt, opt, ...

LSI: List set name table. Outputs set names and values currently active.

LXI: List transfer table. List contents of IDPXFT. Octal DPC.

MUL: Integer multiply. Provides a 60-bit result.

PAS: Parse subexpression. The parse routine for expressions in IDP directives. The expression is parsed (shift reduce) into a polish string, which is then evaluated, returning an absolute value.

PAT: Parse FWA, LWA, LEW triple. PAT makes successive calls to PAS to evaluate the indicated parameters.

PEM: Print Error Message. PEM converts the diagnostic pointers into dictionary words and formats the diagnostic line for output.

POL: Process options list item. POL is called by IDP SC= routine to process a list item. Uses the required subkeyword table for the request.

PTR: Pointer manager. PTR manages the access and updating of user interface pointers.

RIL: Read IDP input line. Under control of status flags, reads IDP directive lines into a buffer for translation.

ROL: Write output line. Utility to provide IDP an output mechanism.

SKT: Search keyword table. SKT is a routine called by CST to actually search the table for a command verb.

SLF: Search for logical file name. Searches for a user supplied LFN through the FET/FIT table and FA.SSW. Returns FWA of FET, if found.

SOB: Set output flag bits. Output bits for batch listing control are set to differentiate between batch and interactive modes.

SSY: Search Symbol Tables. Searches IDP's various symbol tables to associate a binary value with a DPC name. If found, a code is returned denoting type of symbol (deckname, set symbol, user symbol, etc.).

STP: Step an instruction. STP steps a single CPU instruction and lists result register as necessary. Called when step mode is invoked, for each instruction of the step range.

TOGEL/TOK: Token generator for IDP request processing. Contains token skeleton macro calls and token generation routines.

UBK: Unbreak. Removes indicated break and restores user code at that point.

UEQ: User Freeze Owncode. Interface to user provided freeze processing.

UER: User freeze restart owncode. Interface to user provided freeze restart/restore code.

UIQ: User IDP owncode. Interface to user supplied IDP executive.

URQ: User REG owncode. Interface to user supplied register dump routines.

USQ: User SNP owncode. Interface to user supplied snapshot routines. The five user interface routines utilize 'soft' externals and are satisfied either by the user, or by canned IDP routines.

VAR: Process variable token: used by TOGEL/TOK to special process entokening of variable names.

k. BUB: Burst/build characters with blank squeeze. Burst an input line into individual characters, trashing blanks. Used in entokening IDP requests. Comdeck COMCBUB.

l. BUN: Burst characters, no blank squeeze. As BUB, with no characters eliminated. Comdeck COMCBUN.

m. CDQ: Constant to decimal conversion. Comdeck COMCCDD. Described in UTILITY (2.2).

n. CIQ: I/O function processor. Comdeck COMCCIO. Described in UTILITY (2.2).

o. COQ: Convert constant to octal DPC. Comdeck COMCCOD. Described in UTILITY (2.2).

p. DXB: Convert DPC to binary. Comdeck COMCDXB. described in UTILITY (2.2).

q. MCS: Merge Coded Strings. Concatenates zero filled strings into a new string of up to 2 words. Trailing : (64 character set) are lost. Comdeck COMCMCS.

r. RDC: Read Coded line. Comdeck COMCRDC. Described in UTILITY (2.2).

s. RDW: Read words to working buffer. Comdeck COMCRDW. Described in UTILITY (2.2).

t. RDX: Read exit. Comdeck COMCRDW. Described in UTILITY (2.2).

u. LCB: Load Circular Buffer. Comdeck COMCRDW. Described in UTILITY (2.2).

v. RSR: Restore all registers from a specified save area. Comdeck COMCRSR.

w. SBM: Set block of memory. Comdeck COMCSBM. Described in UTILITY (2.2).

x. SFN: Space fill name. Comdeck COMCSFN. Described in UTILITY (2.2).

y. SVR: Save registers in a specifier register save area. Comdeck COMCSVR.

z. SYS: Process system request. Comdeck COMCSYS. Described in UTILITY (2.2).

aa. COMCIOK: Routines and structure used in entoken IDP requests. Described in LEX (3.3).

bb. WOD: Convert word to octal DPC. Comdeck COMCWOD. Described in UTILITY (2.2).

cc. WTC: Write coded line. Comdeck COMCWTC. Described in UTILITY (2.2).

dd. WTW: Write words to working buffer. Comdeck COMCWTW. Described in UTILITY (2.2)

ee. WTX: Write exit. Comdeck COMCWTW. Described in UTILITY (2.2)

ff. DCB: Dump circular buffer. Comdeck COMCWTW. Described in UTILITY (2.2).

gg. XJR: Restore all registers with a system XJR call. Restore based on a full word save area (for each register). Comdeck COMCXJR.

hh. ZTB: Zeros to blanks. Comdeck COMCZTB. Described in UTILITY (2.2)

NOTE: Where the comdecks/routines were marked as described elsewhere, IDP required the indicated function. But at any given time a break may occur in those same routines; thus, IDP keeps local copies of the necessary routines.

2.9 Initialization Routines

For each overlay loaded (primary or secondary), there is a corresponding initialization routine. These routines vary in size and function, but in each case the initialization is performed once, when the overlay is loaded, and then the space becomes available for managed table, scratch storage, etc. These routines are not true cradle (a separate routine exists for each overlay), but, since every overlay has one, they appear in the cradle grouping.

2.9.1 INITOO: Primary Initialization

Abstract: INITOO provides primary, first time only, initialization for FTN5. The initializations provided mostly pertain to options present (or absent) from the control statement.

Interfaces: INITOO interfaces the cells and data structures of FTN. It is contained in overlay (0,0) and contains the compiler main entry point. As such, INITOO is always invoked once. After performing the required initializations, the space occupied by INITOO becomes free space.

Data Structures

a. COMECIP: Installation parameter. Described in FTN (2.1).

b. COMCPAC: Several structures are defined for use by PAC during control statement parameter processing.

   Character Mapping: An entokening of the control statement parameters.

   KA. KB. KC: Parameter table definitions.

   KD. KE: Multiple binary parameter table definitions.

c. KEYS: Control statement parameter keyword table. Expansions of PARAM macro (COMCPAC) in format determined by KA., KB., KC. definitions.

d. MBV: Actually a series of multiple binary value tables. Expansions of MBVOP macro (COMCPAC) in format determined by KD. and KE. definitions.

e. CS cells: Local cells used in processing control statement parameters. These provide temps for options which require 3-way action.

f. Diagnostic Texts: Both PAC and INITOO provide tables of diagnostics text and exit addresses.

g. SK.: Code skeleton data definition. Used by ROR. Described in FSKEL (3.16). Comdeck COMSEIS.

a. EIN: The system loader entry point. Calls the initialization routines and, after determining the control statement parameters, either transfers control to PUC or calls the overlay loader to load (1,0) [OPT=0, MAP or LIST needed], (2,0) [OPT>0] or COMPASS [IDENT first card].

b. PAC: Process arguments from control statement. PAC is the control statement cracker. It processes each control statement argument in turn, scanning against KEYS for legality of the argument. The corresponding KEYS entry is used to test the argument value and errors are noted, or the proper cells are set, corresponding to the value supplied. In the case of multiple binary, the proper MVD table is scanned for syntax legality. Any errors noted by PAC result in termination of compilation. Comdeck COMCPAC. One option of PAC processing is the provision for user routines. INITOO provides routines to process the S and G options, and in test mode, the IDP and SNAP options.

c. CFL: Check Field Length. CFL determines the current field length and, if above the minimum, exits. If not, the nominal field length (for the OPT level in force) is substituted and a MEM request is made. If the MEM cannot be honored, compilation is terminated.

d. CFN: Change file name. CFN changes the default file vector name to that name requested by the control statement, or clears the vector if 'file'=0 was specified.

e. MIA: Miscellaneous initializations - A. MIA provides initializations which are performed for all compilations, regardless of control statement parameters. Start time is saved, compiler status noted (system vs. library or file), field length is saved, etc.

f. MIB: Miscellaneous initializations - B. MIA does the bulk of INITOO initialization. It is called after the control statement has been successfully cracked, and resolves irregularities in the control statement requests. For some options, the control cell contents are converted to compiler usable form. (Some options have assembled values in a form which is different from the PAC converted form. This is to differentiate between a selection of the default value and not selecting the value but assuming the default.) All dependencies between control statement parameters are resolved. Any diagnostics noted by MIA results in termination of compilation.

g. PPW: Process page width option. The output files' (OUTPUT and ERROR) page width is set depending upon the PW option status and the output media (printer, TTY, terminal, etc.).

h. CPV: Current parameter values. Converts values of selected control statement parameters into display information for the header information of listings.

i. FCA: Find character-address. Utility used in processing control statement line for listing header.

j. TCC: Transfer continuation control card. Utility used in producing control statement line for listing header.

k. STF: Set Terminal File. STF determines if a file is assigned to an interactive device. Comdeck COMCSTF.

l. GOI: Global Overlay Initialization. GOI performs initialization required by the QCG mode code generator. If necessary, the code for map/list is trashed. Routines are called to perform front end initializations. Comdeck COMFGOI.

m. FEI: Front End Initialization. FEI initializes flags and switches in front end routines (e.g., ANSI diagnostic switch). In (0,0) overlay, a QCG function. Comdeck COMFFEI.

n. ROR: Reset opcodes of roundables. Depending on the status of the ROUND option from the control statement, ROR resets the affected code skeletons to rounded/unrounded arithmetic. Comdeck COMFROR.

2.9.2 INIT1O: Reinitialization (GCG)

Abstract: INIT1O performs initializations required when FTN5 is reloaded after an intermixed COMPASS assembly, or when MAP or LIST was required, from the initial load.

Interfaces: INIT1O contains the loader entry point for the (1,0) overlay. Resides on the (1,0) overlay.

Data Structures

SK.: Code skeleton data definition. Comdeck COMSEIS. Described in FSKEL (3.16).

Routine Descriptions

a. EIN1O: The overlay entry point. Calls initialization routines and exits to PUC.

b. GOI: Global Overlay Initialization. Comdeck COMFGOI. Described in INITOO (2.9.1).

c. EEI: Front End Initialization. Comdeck COMFFEI. Described in INITOO (2.9.1).

d. ROR: Reset opcodes of roundables. Comdeck COMFROR. Described in INITOO (2.9.1).

2.9.3 INIT20: Initialize Primary Overlay (CCG)

Abstract: INIT20 performs initializations required when the CCG primary overlay is loaded.

Interfaces: INIT20 contains the loader entry point for the (2,0) overlay. Resides on the (2,0) overlay.

Data Structures

COMECIP. Comdeck, described in FTN (2.1).

Routine Descriptions

EIN20: Entry point for the (2,0) overlay. Entered when OPT>0 after INIT00 initialization, or upon return from intermixed COMPASS ASSEMBLY. Saves initial field length and exits to PUC.

2.9.4 INIT21: Front End Initialization (CCG)

Abstract: INIT21 performs initializations required by the (2,1) overlay, the CCG front end processor.

Interfaces: INIT21 contains the loader entry point for the (2,1) overlay.

Data Structures

SK.: Code skeleton data definition. Comdeck COMSEIS. Described in FSKEL (3.16).

Routine Descriptions

a. EIN21: The overlay entry point. Calls the initialization routines to perform front end initializations. Exits to the front end controller.

b. DLE: Dump link and fill tables. A front end stub for read end function.

c. EEI: Front end initialization. Comdeck COMFFEI. Described in INITOO (2.9.1).

d. RQR: Reset opcodes of roundables. Comdeck COMFROR. Described in INITOO (2.9.1).

2.9.5 INIT22: CCG Initialization

J3

Abstract: INIT22 performs initializations required by the common code generator.

Interfaces: INIT22 contains the loader entry point for the (2,2) overlay.

Data Structures

a. SK: Code skeleton data definition. A CCG version of the SK. defined for OPT=0. Used by ROR.

b. Table Definitions: Comdecks COMSTAB, COMSTAD and COMSTAS. Shared table definitions for CCG used. Described in PUC (2.3).

Routine Descriptions

a. EIN22: The overlay entry point. FTN22 performs some file management functions and initializes CCG cells. The managed table area is restructured for CCG use. Exit is to the CCG controller.

b. ROR: Reset opcodes of roundables. Comdeck COMFROR. Described in INIT00 (2.9.1)

J4

2.9.6 INIT23: Rear End Initialization (CCG)

Abstract: INIT23 performs initializations required by the (2,3) overlay, the CCG rear end processor.

Interfaces: INIT23 contains the loader entry point for the (2,3) overlay.

Data Structures: None

Routine Description

EIN23: The overlay entry point. Performs some file management functions for the map and listing functions as required. Exits to the rear end controller.

3.0 FRONT_END_PROCESSOR

The FTN5 front end processor consists of decks and routines to provide lexical analysis, parsing and intermediate language production. A symbol table and other auxiliary tables are built. The intermediate language drives both the QCG and CCG code generators.

3.1 FEC: Front End Controller

J6

Abstract: FEC controls processing flow for the front end routines. Functions include calling the lexical analyzer, transferring to the statement processor and inter-statement initialization. FEC also contains a number of routines used by the front end for various functions.

Interfaces: FEC is the traffic director and controller for the front end. Contains program unit and individual statement flags and cells. FEC resides on the (0,0), (1,0) and (2,1) overlays.

Data Structures

a. Cells: FEC cells are entries which are used by the front end processor. They contain information needed by the front end for bookkeeping functions. The cells area, in conjunction with the managed table area and the program unit cell area, define the status of compilation at any given point.

b. F.SYMIL: Symbol table initialization table. F.SYMIL contains the symbol table preset values for the compiler generated, fixed ordinal symbol table entries.

c. HASHTBL: A set of hash buckets, used by the symbol table accessing routines. Contained in common block /HASH/.

d. CHARMAP: CHARMAP is a table used for various purposes. It is ordered as the O. table (described in FTNSTXT (1.1)). The table consists of DPC representation of tokens/operators and a DUC. address, where applicable, for use by QCG. CHARMAP is used for diagnostic production, QCG transposition and some output functions.

e. FEC=: FEC= is the stage vector for the transition between the lexical analyzer and the various statement processors. The table consists of a matrix of current (just entokened) statement type by current program unit stage. The entries are offsets into the FEC main loop.

Routine Descriptions

a. FEC: Front End Controller. FEC is the front end main loop. Upon first entry for the program unit, initialization routines are called to set up compilation. Thereafter, FEC provides entry points for return from the various statement processors and stage processors for lexical to syntatic transistion. First the statement specific cells are reset, then LEX is called to entoken the current statement. Based upon values returned by LEX, pertaining to the type of statement just entokened, a stage vector branch is taken to process the statement context. The stage transform routines call routines to perform the necessary processing where required. If statements appear out of context (e.g., type declaration after first declarative), a diagnostic is produced. After the stage transformation processing is completed, transfer is to the relevant statement processor. A special return is provided for the END statement. Front end processing is completed by a series of subroutines. Transfer is to the front end loader to fetch in the code generation overlay (CCG) or to transfer to rear end processing (QCG).

b. ASK: Adjust statement keyword. Some of the syntax of FORTRAN is 'funny' (i.e., doesn't lend itself to lexical analysis nicely). Situations where consecutive keywords (implicit type declarations) or imbedded keywords (ASSIGN) are present require this routine. ASK strips out the O.VAR token containing the relevant keyword of the keyword characters. The relevant token is discarded or adjusted (if information remains). This adjustment may involve retyping a token.

c. ASL: Adjust Statement Label. Similar to ASK, except that a statement label is extracted from the token. Used by statements where a label is followed by undelimited information (ASSIGN and DO).

d. CAC: Check assumed character declarations. A front end cleanup routine. When character type variables have been declared, CAC scans the symbol table to determine if any variables have been invalidly declared to be of assumed length.

e. CBN: Check common block names. Searches symbol table for non-ANSI usage of common block names. Conflicts are diagnosed. Called only when ANSI is specified on the control statement.

f. CLU: Check Level Usage. A front end cleanup routine. If LEVEL was declared, CLU scans the symbol table, testing for leveled items being in common or formal parameters. Local variables leveled are diagnosed and LEVEL 0 formal parameters are marked in the symbol table.

g. CSB: Check Sequence Break. CSB is called when a possible IL sequence break can occur. If a sequence break has occurred (subroutine call, some labels, long parse file), the IL is flushed. Action is dependent upon the code generation mode.

h. CUF: Check Undefined Function. A front end cleanup routine. CUF determines if a function subprogram has been defined. If entry points of more than one type were present, CUF determines that the entries were defined. If not, a diagnostic is issued.

i. CUL: Check Undefined Labels. A front end cleanup routine. CUL scans the symbol table for undefined (but referenced) statement labels. A diagnostic for each missing label is output. If the block structure table is not empty, its contents are analyzed and diagnostics for unclosed do loops and/or unterminated IF blocks are output.

j. CUS: Check Upcoming Statement. For control flow statements (e.g., GOTO, IF), some front end optimizations can be performed if the next statement is known. Thus, the concept of the 'hanger' processing. When a control transfer statement is appended to a logical IF (or an arithmetic IF is found), generation of the branch turples is deferred until the next statement is entokened. If the next statement is labeled, and the label is significant (is part of the branch statement), more efficient code can be generated. CUS determines if 'hanger' processing is required and, if so, transfers to the proper routine for processing. CUS calls CSB for sequence break check and determines if a no path situation exists, is continued or was terminated.

k. CVD: Check Variable Dimension irregularities. A front end cleanup routine. If variable dimensions were declared, CVD scans the symbol table to determine that variables used as the adjustable dimensions were declared to be common or were formal parameters and that arrays with adjustable or assumed dimensionality were formal parameters.

l. FEP: Front End Presets. FEP is called for each program unit to provide initialization of FEC cells and tables.

m. FVD: Flush Variable Dimension code. A front end cleanup routine. FVD will take turples, if present, from the variable dimension table and move them to the parse file. Thus, any code generated for variable dimension code will follow the body of code for the program unit.

n. OIL: Output intermediate language. OIL is used to flush the parse file. Unless code generation has been supressed, the parse file is flushed, via a call to PIS. Manner of flushing is code generation dependent.

o. PUP: Program Unit Presets. PUP is called for each program unit to initialize PUC cells and tables. This initialization could have been placed in PUC, but inclusion in FEC allows it to 'go away' when the CCG and rear end overlays are fetched during CCG compilation.

p. RLS: Relocate local save variables. A front end cleanup routine. If variables were declared in a SAVE statement, RLS will create a special local block 'SAV$E' in which local save variables will be located. This will allow special rear end processing of saved variables. The block table is updated to include SAV$E.

q. RSC: Reset intrastatement cells. RSC is called once for each FORTRAN statement encountered. It resets values of FEC flags and cells which pertain to the status of an individual statement.

r. SSU: Set Save Universal. When an unqualified SAVE was declared, SSU will scan the symbol table, inserting the save bit in all octal and common variables. A front end cleanup routine.

s. BBC: Base bias conversion. A front end support routine. BBC determines the need for the conversion (equivalenced variable) and if necessary, replaces the operand with one consisting of the base member symbol as the ordinal, the offset from the base member as the bias and the mode of the original operand.

t. CCT: Check conflicting types. A front end support routine. CCT tests a proposed symbol table class bit against a list of forbidden classes. If the proposed class is forbidden, the DPC for that class is determined and used as fill for a diagnostic which is output.

u. CT1: Construct pass one tag (archaic terminology). A front end support routine. CT1 takes a symbol table ordinal as input, fetches the corresponding symbol table entry and forms a TP. format operand, utilizing and transforming the WB. symbol table information into TP. information.

v. STY: Set natural type. STY is called to provide the implicit type of defined by usage variables. STY has two associated tables NAT.LEN and NAT.TYP. First, the initial character of the variable (or subprogram name) is used as a shift count for a bit check of NAT.TYP, an array of bit vectors. This determines the implicit type. If the implicit type is character, the initial character is used as an index into NAT.LEN, a table of implicit character lengths. A front end support routine.

w. TLV: Truncate long variable. A front end support routine. TLV is called by various statement processors when consecutive variable tokens (O.VAR) are encountered. The first token is shifted over the last variable token of the string and the token buffer pointer is reset. A diagnostic is output.

x. TRV: Translate variable. A front end support routine. TRV is called when a variable symbol is expected. The variable token currently in the token buffer is searched for in the symbol table. If not present, an entry is made, with the proper variable attributes. If a symbol table entry exists, the attributes are checked and if an irregularity occurred, a diagnostic is output. TRV returns the symbol table WB. entry, a TP. operand and the symbol table ordinal of the variable.

y. TSX: Tag system external. A front end support routine. TSX is called to enter the name of a compiler generated subprogram (library routines) into the symbol table. Returns the WB. entry, a TP. operand and the symbol table ordinal.

z. TSY: Tag compiler symbol. TSY is a specialized routine which is called to generate symbol table entries for certain unique compiler generated symbols. An error results if attempt is made to enter a symbol more than once.

aa. ERT: Enter Reference Table. A front end support routine. ERT is called to format symbol references for the cross reference map. If LO=R is not specified, the initialization routine for the current overlay will wire off ERT. Once the switch is set on, a subsequent C$ LIST R= directive can set or unset the wire off code. Once entered, ERT will combine symbol table ordinal, line number and current usage into an entry for the cross reference table and will add the entry to the table (or file if references have been dumped to disk).

bb. ESY: Enter symbol table. A front end support routine. ESY enters a symbol and its attributes into the symbol table. The hash link chain is updated to include the new symbol. ESY requires a previous call to SSY for linkage (symbol table hash chain) reasons.

cc. INN: Invent New Name. A front end support routine. FTN5 has several requirements for compiler generated names. Some of these are compilation dependent and can occur many times during a program unit (e.g., DO loop variables and labels). For this class of symbol, INN will take a count of the number of occurrences of the symbol class, add a prefix and make a unique symbol table entry for the compiler generated symbol.

dd. NCM: Enter multiword element into designated table. A front end support routine. NCM scans a table searching for a match with an arbitrary number of words (to be entered in the table). If a match is found, the table index of the first word of the matching block is returned. If the block is not in the table, the necessary space is allocated and the block entered. This last is at he caller's discretion. (Some applications allocate space, build a block and call NCM. If the block was redundant, the table is shrunk; otherwise, the data is already in place.)

ee. SCS: Scan table with supplied mask. SCS searches a given table for an entry which matches at only those bits denoted by the supplied mask. (As opposed to a 60 bit match). Comdeck COMFSCS.

ff. SCT: Scan table comparing all bits. A front end support routine. SCT scans a designated table, looking for a provided entry. If a match is found, the index of the matching entry is returned; otherwise, a miss indication is returned.

gg. SLT: Scan Library Table. SLT scans the intrinsic function table (F.INTF) for a function name supplied. If the desired function is in F.INTF, symbol table attributes are extracted from that table and put in WB. and WC. formats. Otherwise, the attributes returned are those of a user function.

hh. SSY: Scan symbol table. A front end support routine. SSY accepts the DPC for a symbol and applies the FTN5 hash function to yield the symbol hash value. This value is used to scan the relevant hash chain to determine if the symbol is present. If present, SSY returns the symbol table ordinal and index and the WB. entry for the symbol. If the symbol is not in the table, a pointer to the last link pointer (end of chain) is set and an indicator of not in table is returned. SSY must be called prior to entering any symbol in the symbol table. (Note: The hash function used by FTN5 was lifted from FTN4 which, in turn, used the COMPASS hash function. Since COMPASS uses a different alignment for symbols [OR vs. OL], it is not clear that the current algorithm is best for FTN5. If some testing is done someday, this could be determined. For now, collisions don't seem super rampant, so nothing is pressing about this.)

3.2 FERRS: Front End Diagnostic Texts

Abstract: FERRS contains the texts of all diagnostics output by the FTN5 compiler. A few diagnostics which may occur during rear end processing are included for (0,0), (1,0) compatibility.

Interfaces: FERRS interfaces all decks/routines in the front end which issue diagnostics. A diagnostic output consists of a branch to the diagnostic entry in FERRS followed by a branch to the proper formatting routine in PEM (2.5). FERRS resides on the (0,0), (1,0) and (2,1) overlays.

Data Structures

FERRS consists of two main data structures, diagnostic texts and dictionary, with associated macros for their generation.

a. COMSSYC: This comdeck provides macros and micros to generate literal DPC values associated with symbol class attribute bits. The generation of the literals is remote.

b. COMAERR: This comdeck provides the ERROR macro, used to format the diagnostic texts. Format for the macro is:

```
LOC ERROR TYPE, EXIT, (TEXT)
```

where

LOC - diagnostic name and entry point

TYPE - severity and what type formatting PEM is to perform

EXIT - return address or default return

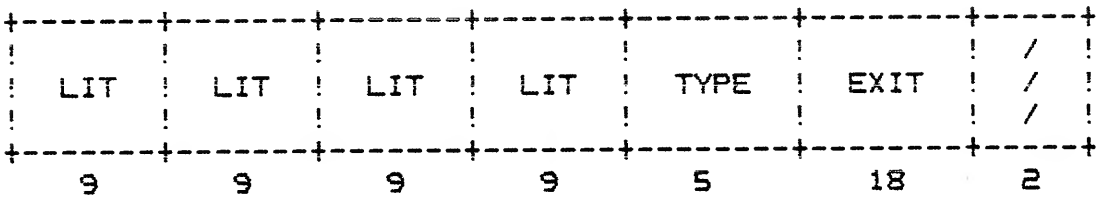
TEXT - the actual diagnostic text.

The error macro will generate the following code:

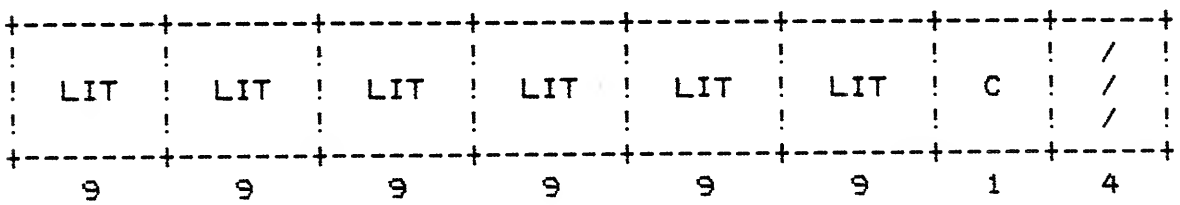
```
LOC SB7 return address
    EQ  PEM    S
               V
               ⊻    depending on the PEM action required
                    (the second character in the type
                    field, if present).
```

Following the entry code are the words making up the diagnostic skeleton. The format is as follows:

1ST_WORD



2ND_AND_SUBSEQUENT_WORDS



where:

LIT - Offset into literal table (dictionary). Note that the size of the LIT field limits the dictionary to 512 entries. Some care in wording new diagnostics will keep the dictionary within this range.

TYPE - The diagnostic severity level. (EL=level)

EXIT - Diagnostic return address.

C - Set if further skeleton word(s) exist.

Dictionary_Format_and_Production: Dictionary entries are produced by the ERRLIT macro, which counts the number of characters in the proposed entry and then blank pads (left-justified) and if <10 characters, appends a special count character as the 10th character. If the dictionary word is exactly 10 characters, nothing is done and if >10 characters, the 10th character is replaced by blank (55B) to indicate continuation and a second dictionary word is formed with the remaining characters.

COMAERR defines the entry points for and ERRLITs of the first elements of the dictionary. These include a blank word and the FILL. cells. The FILL. cells are preset with the literal 'FILL.' so that the LIT pointers in the diagnostic skeletons will be set up properly, but the cells are then used for variable information, set by the caller, interpreted by PEM.

4. Generate a string of tokens for each source statement.

5. Determine which statement processor is to process each statement (i.e. type each statement).

Figure 3.1 shows the relationship of the scanner to the whole.

---

* Source lines that are found to be in error in NO LIST mode (L=0, SL=0, or C$ LIST NONE) active) are listed by the compiler's error processor, PEM (Print Error Message) in deck PEM.

** Alternately, this can be seen as collecting lines into groups called statements.

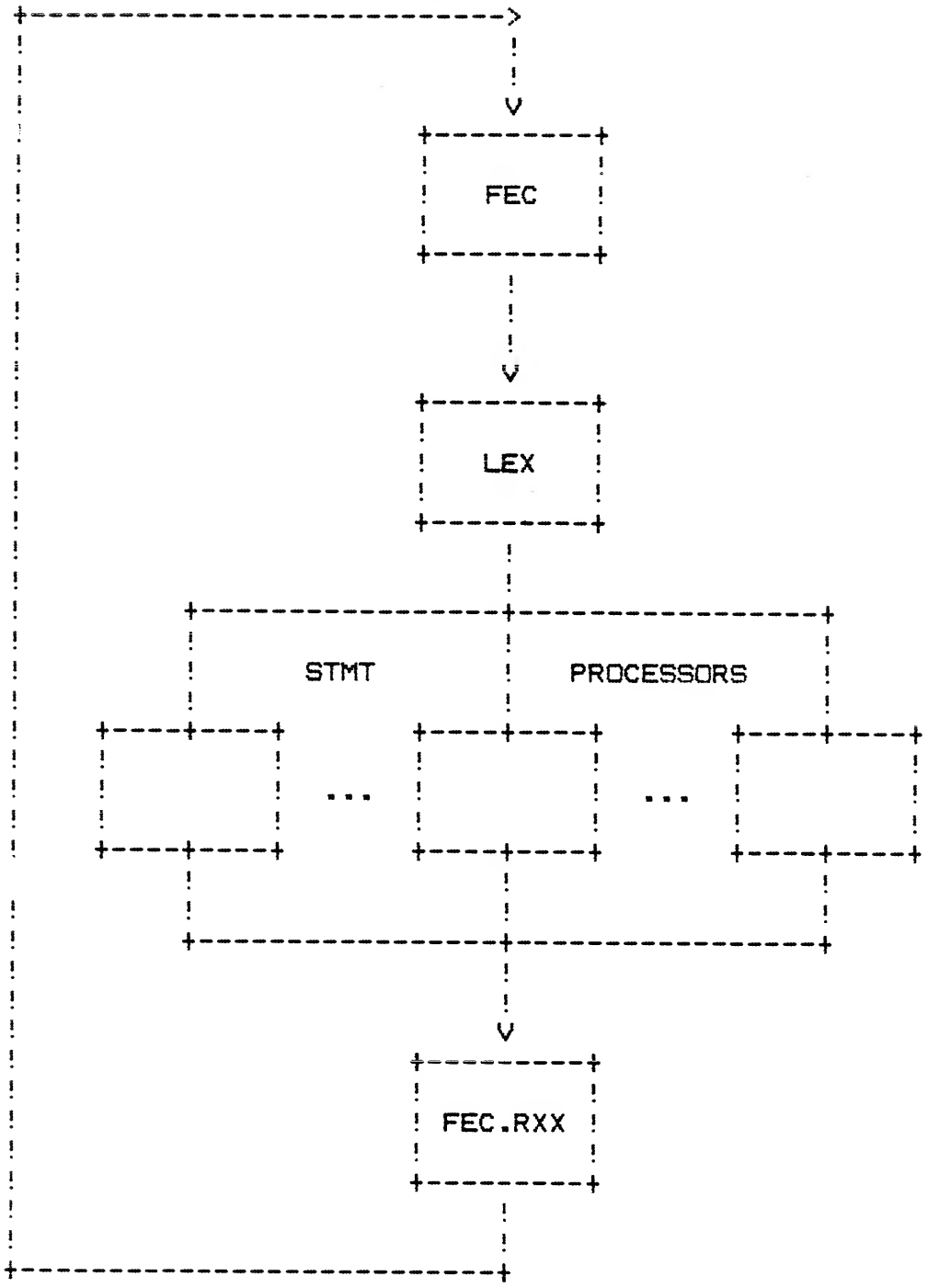


Fig. 3.1. Simplified front-end master loop with LEX

Upon entry, LEX will read, list, and entoken source lines until it detects a line that is the beginning of the next statement (remember that in FORTRAN, the only way to detect end of current statement is to detect beginning of next statement). Control then returns to the compiler master loop, where the appropriate statement processor is invoked to compile the statement.

This process repeats itself for each source statement until the end of the program unit is detected.

3.3.1.2 Interfaces

LEX primarily "talks" to the rest of the compiler via its data structures. These interfaces are, for the most part, predetermined by the FTN 5 overall design, and are inviolate as far as the scanner is concerned. It is hoped that by stating these things first, the hows and whys and wheres will be easier to understand and criticize.

This section is divided into 2 sub-sections: INPUTS and OUTPUTS.

a. INPUTS

Because the scanner is so close to the beginning of the compilation process for a source statement, its inputs are relatively simple and straightforward. It expects some data cells to be initialized (usually just cleared), and for the 1st time it is called, it expects the INPUT and OUTPUT files to be initialized*. In addition, it references a number of global flags, most of which contain control card information (for example, does the programmer desire a source listing, etc.). The INPUT file itself is an input, but because the scanner manages it for the most part without interference from the rest of the compiler, it is in a slightly different class. See figure 3.2.

---

* MIB (Miscellaneous Initialization, Part B) in deck INITOO sets up the file FETs (or pseudo FETs in a RECORD MANAGER world), OPENs the INPUT and OUTPUT files, and performs the initial READ to fill the INPUT buffer.

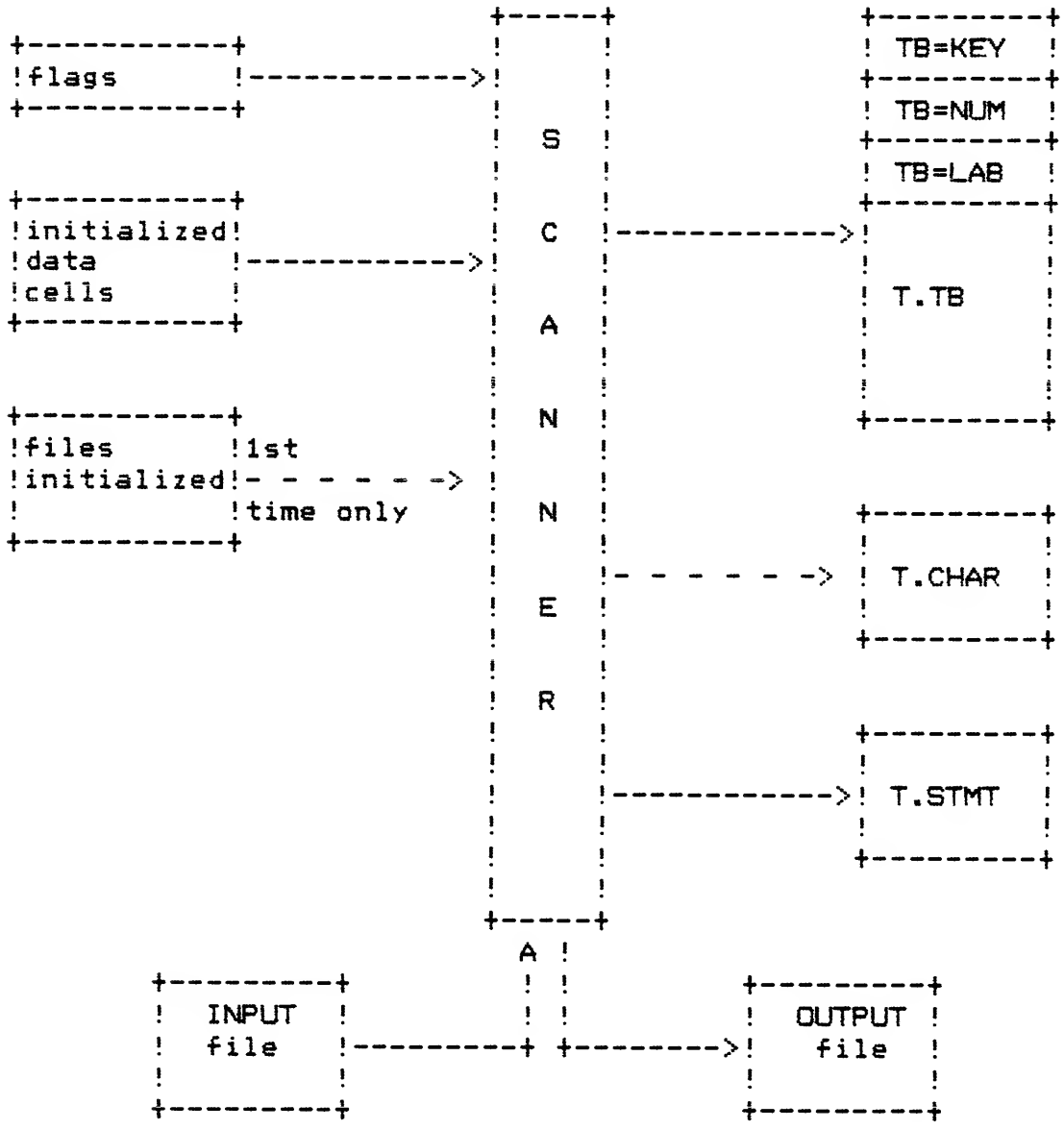


Fig. 3.2. Scanner inputs and outputs

b. OUTPUTS

The scanner's outputs are confined to the listable output file, a number of miscellaneous data cells and flags, and to three managed tables. Each of these managed tables is described below in a somewhat general fashion. More detailed information can be found in subsequent areas that deal with them.

T.TB

This table is the token buffer and is generated for each and every source statement. At any one time, T.TB only contains the tokened representation of a single source statement. This "entokened statement" is always preceded by a BOS (beginning of statement) token, and is always terminated by an EOS (end of statement) token.

Associated with T.TB are a number of data cells which are all related via the common prefix "TB=". These "TB=" cells contain information about the current statement in T.TB.

For example, LEX stores information about which statement processor is to compile this statement in the cell TB=TYPE.

The cells TB=NUML and TB=NUMR are set to contain the line number of the 1st line of the statement in T.TB (in left and right justified form, respectively). This source line number is passed to FTN object library subroutines so that if an error is detected, the programmer can be informed as to which source line is at fault. For SEQ mode input programs, the SEQ line number is used.

The cells TB=LABL and TB=LABR (left and right justified, again) are set to the statement label (if there is one) for the statement in T.TB.

See the actual definitions in the deck LEX for a more thorough description of all the "TB=" cells.

T.CON

This table is an auxiliary to T.TB and contains any character constant strings that occur in a statement. If a character constant does occur, a single character constant token will be generated to T.TB, and this will point to the actual character string in T.CON.

Keeping the actual character strings out of the token buffer tends to simplify the parsing and code generation of character constants, but is not a particularly necessary complication of the token structure. Its main advantage is that T.CON represents a near exact image of what will eventually go into the object code for these character constants. This means that this data does not need to go through any more transformations during the compilation, and can be basically copied to the LGO binary output file.

In addition, a simple optimization can be performed during generation of T.CON. Redundant character constants are ignored via standard table manager protocol (i.e. they just aren't put into T.CON if they are already there).

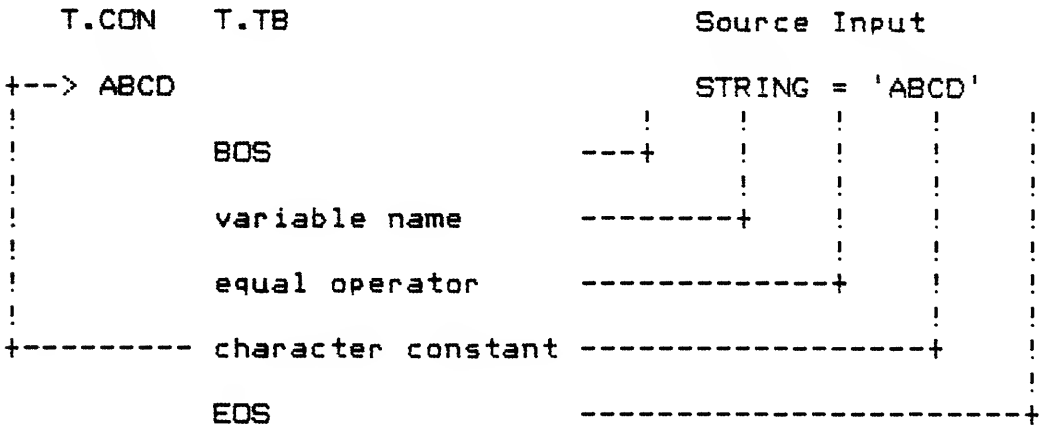


Fig. 3.3. Relationship of T.CON to T.TB

I.SIMI

This table is unconditionally generated for each source statement and contains the statement in its original packed (i.e. 10 character per word) form. Its most obvious use is that of deferred listing buffer. If a statement is found to be in error by the compiler in NO LIST mode (L=0, SL=0, or C$ LIST NONE active), then the statement is unconditionally listed from T.STMT by the compiler's error processor, PEM (Print Error Message) in deck PEM.

T.STMT is also used to accumulate source lines at the beginning of a program unit that are to be listed, but which need to be held until the header statement processor has extracted the program unit name from T.TB and stuffed it into the source listing title line.

The two aforementioned uses of T.STMT are not new, being performed by the deferred listing buffer in both FTN 4 compilers. The third use of T.STMT is new, however, and involves the proposed solution to the FORMAT problem. It is best shown with an example:

```
100   FORMAT
     + (1) = 2
```

The scanner will generate FORMAT tokens for this statement, and upon finishing, will query its heuristic flags. This will lead to the discovery that this is indeed not a FORMAT statement. The scanner now needs to generate normal tokens for this replacement statement. Enter T.STMT. The entokener has to use T.STMT as its source input because it is not practical to try and "back up" the INPUT file to the beginning of the statement*.

---

*See Design/Executives/TOK and Token Generation.

3.3.1.3 LEX Main Loop Concepts

This section is divided into 2 sub-sections. The first describes the major functional components of LEX, and the second is an introduction to the timing of LEX's main loop . . . a sort of "what happens when" discussion.

a. MAIN_LOOP_COMPONENTS

The scanner's main functional components and the name of the sub-executive that oversees each component is represented in Figure 3.4.

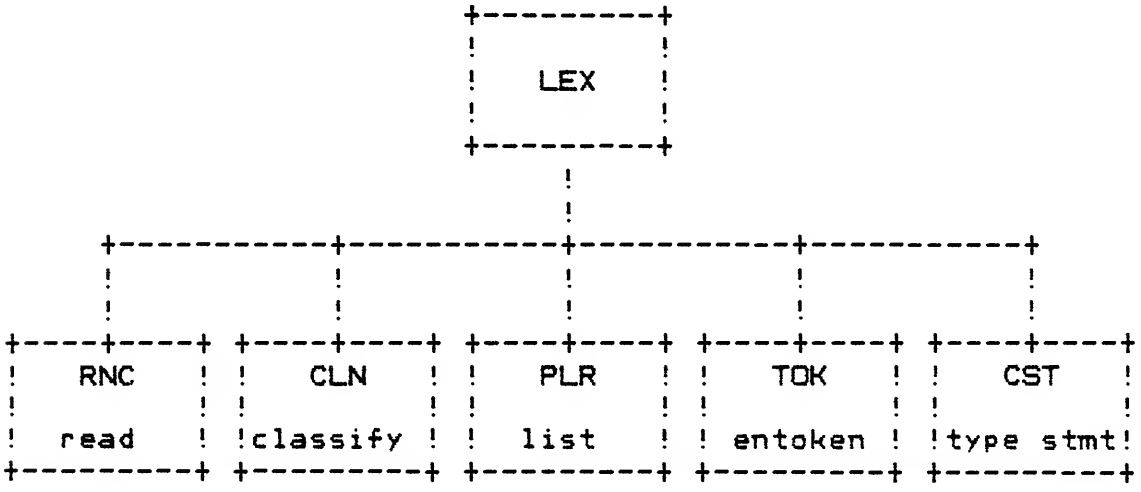


Fig. 3.4. LEX's main functional components

The first 4 components (RNC, CLN, PLR, and TOK) are each called once per cycle through the scanner's main loop. CST only has to be called once per LEX call.

b. MAIN_LOOP_TIMING

LEX is driven strictly on a line by line basis. One cycle is made through its main loop for each line read, listed, classified*, and entokened.

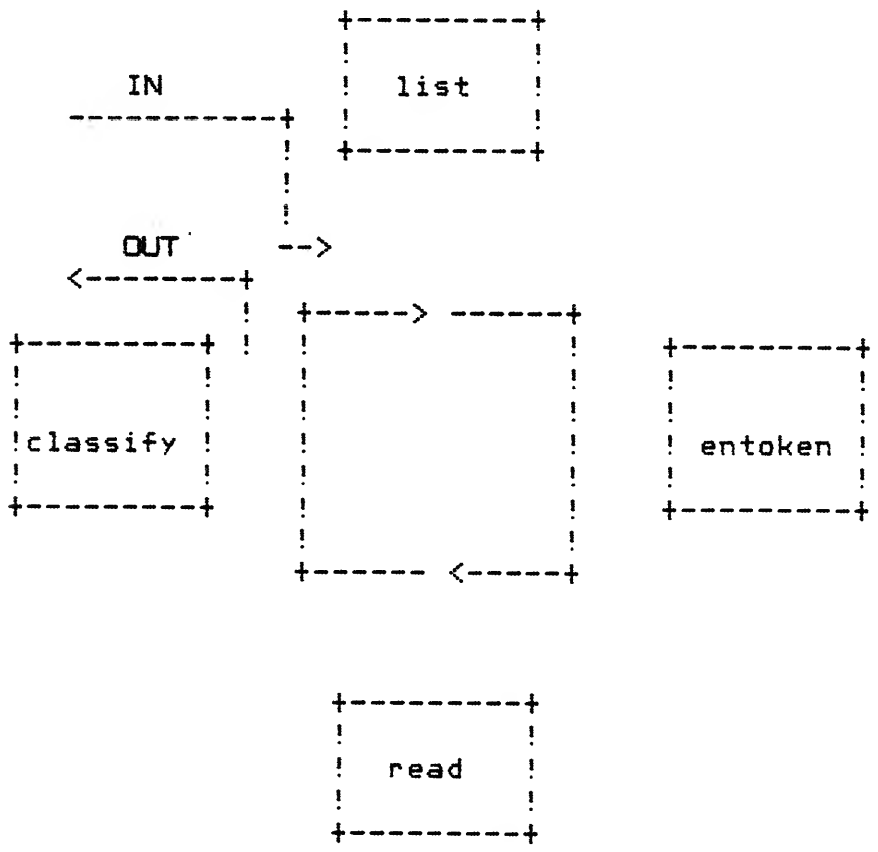


Fig. 3.5. Main loop timing - normal

Figure 3.5 represents the scanner's main loop cycle. IN is when FEC calls the scanner. OUT is when the scanner classifies a line as being an initial (i.e. end of current statement). The OUT path leads to the statement typing mechanisms and then back to FEC to invoke the appropriate statement processor.

---

* Each source line is classified as being one of 5 types: initial, continuation, blank, comment, or C$.

Figure 3.5 also illustrates a very important scanner concept: upon entry at IN, it is assumed that the next line to process/entoken has already been read and classified.

This is true in the general case because this line terminated the last statement, but is not true for a few important exceptions. The most obvious exception is that of the first line of a program unit. This line has to be read before anything else can happen. Figure 3.6 represents an appropriately altered figure 3.5.

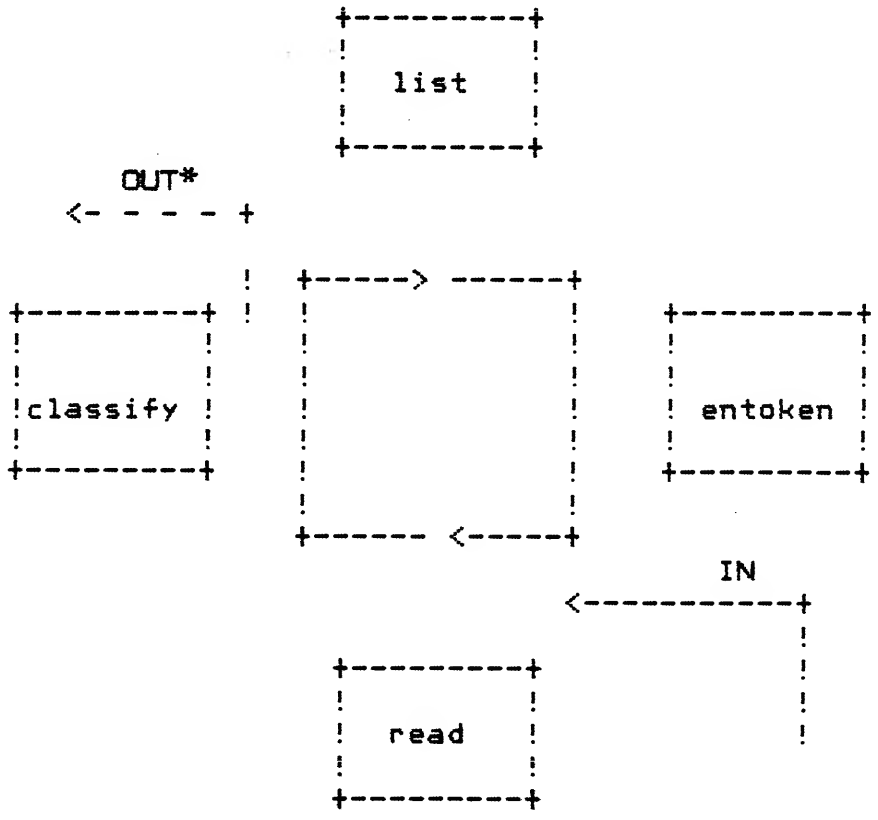


Fig. 3.6. Main loop timing - need to read

This is referred to as the "need to read" case, where the scanner has to read before it can do anything. Currently, the only two conditions where this is necessary are: initializing for the first line of a program unit, and after a C$ directive has been processed.

---

* The OUT path has to be skipped for the 1st cycle through the loop.

** note **

Read-ahead is inhibited when a C$ line is encountered to assure maximum flexibility, because C$ lines can affect line handling logic. See 3.3.3.2 C$ Statement Processing.

***

3.3.2 The Executives

This sub-section has a separate section for each of the executives called by the top-dog, LEX. They are:

3.3.2.1 RNC and Reading

Describes the processes associated with reading source lines. This includes a discussion of general line format, compressed input, etc.

3.3.2.2 CLN and Line Classification

Describes the processes associated with determining what kind of line was just read (i.e. is the line an initial, continuation, comment, null, or C$). This includes a discussion of how SEQ mode input is handled.

3.3.2.3 PLR and Listing

Describes the listing logic and its relationship to the deferred listing buffer, T.STMT.

3.3.2.4 TOK and Token Generation

Describes the token generator as an interpreter of a TOGEL program.

3.3.2.5 CST and Statement Classification

Describes the statement typing mechanisms.

3.3.2.1 RNC and Reading

On operating systems that support CIO, all compiler I/O is done via the MACE I/O comdecks. When CIO is not available, RECORD MANAGER is used via the FA= comdecks*. In either case, RNC (Read Next Card(is the top executive. It calls the appropriate low level executive (via the READC macro) which will read a single source line to the source line image area, CP.CARD**.

---

* The FA= comdecks simulate CIO in a RECORD MANAGER environment.

** CP.CARD resides in COMPCOM in the (0,0) overlay (in deck FTN). See DATA STRUCTURES/CP.CARD.

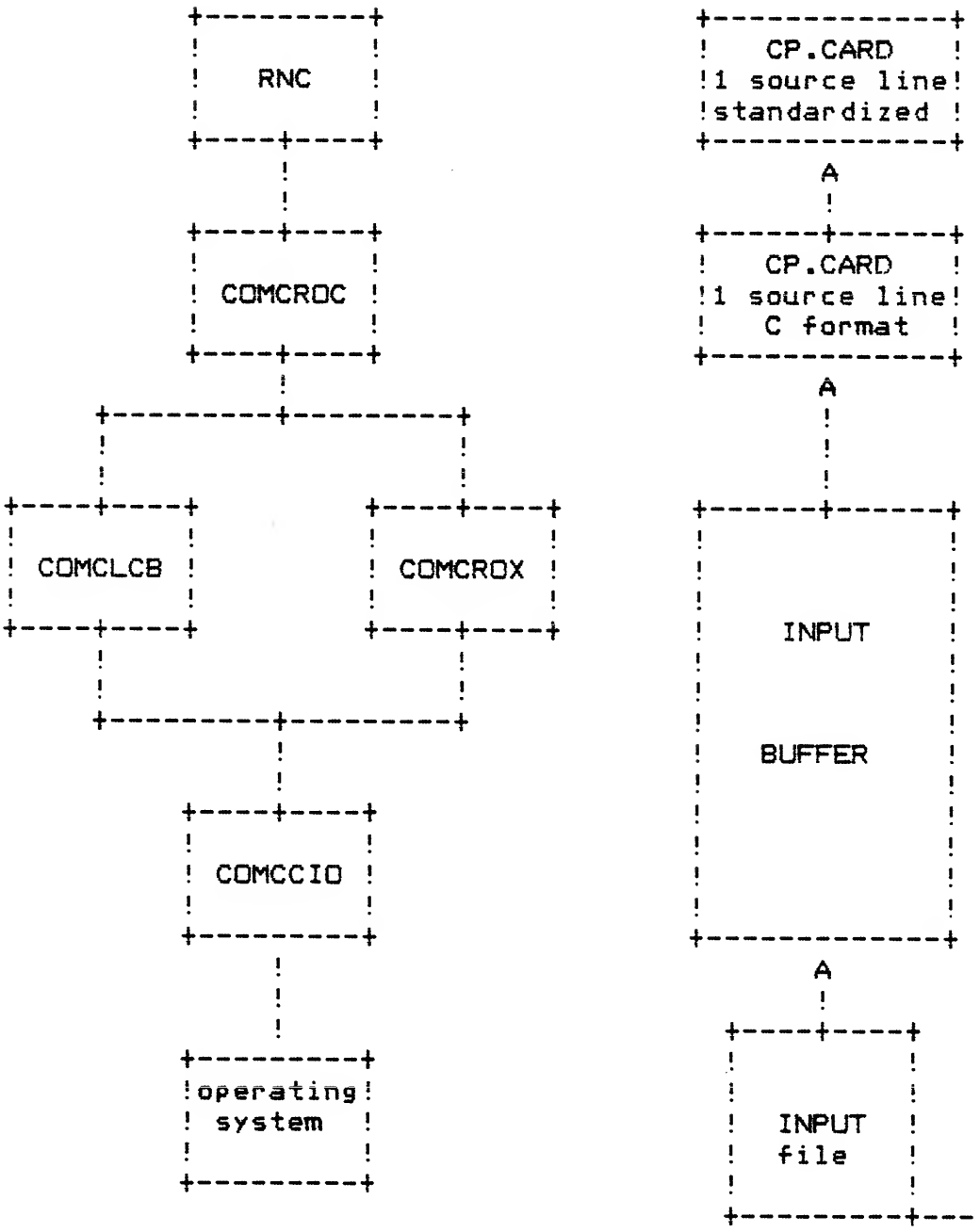


Fig. 3.7. Read routines hierarchy (CIO)

Upon returning from the low level read subroutines, RNC assures that each source line is in a standard format. Standardizing each source line early in the game significantly simplifies line handling logic, for then the scanner does not have to have special case code all over the place to handle the "non-standard" circumstances.

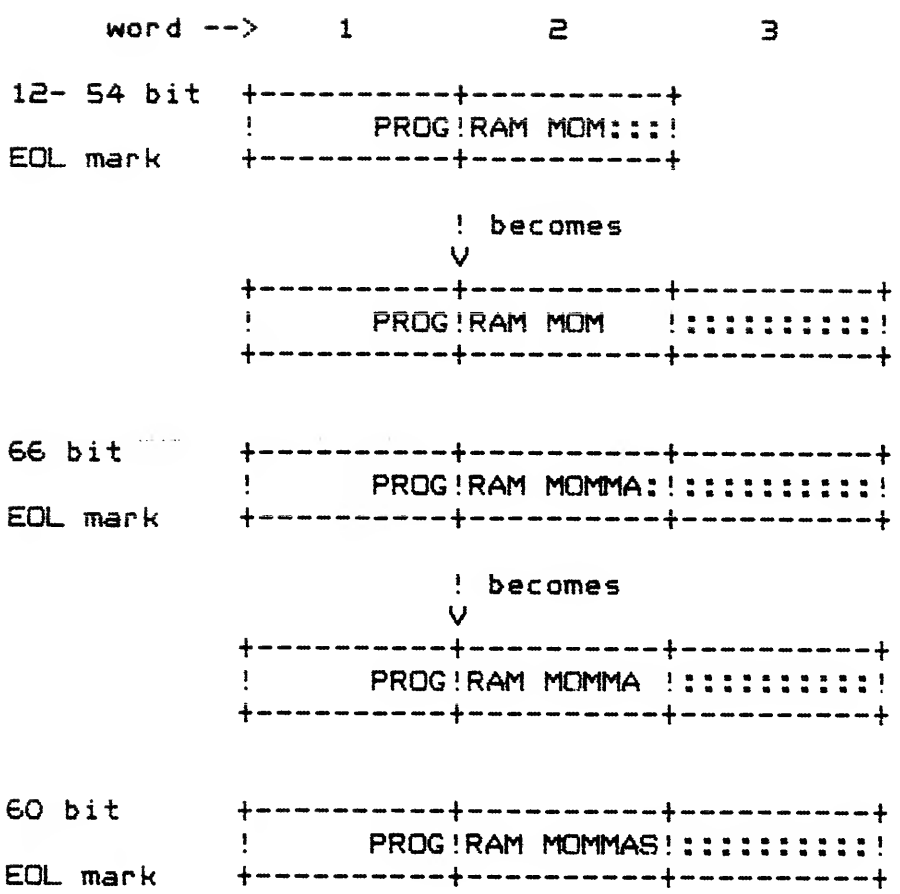
A standardized source line has three primary qualities. First, it is in its packed (i.e. 10 character per word) form. The source line is therefore in a listable form, so that the listing subroutines can list directly from CP.CARD. If we are supporting compressed input for FTN 5, RNC will expand the compressed source line back to its 10 character per word form. Expanding compressed input might seem to be defeating its purpose, but the advantages of standardization far outweigh the de-optimization of expansion. Compressed input will still be a viable optimization for a user because it cuts down the I/O time used by reducing operating system requests. Another advantage to expanding here is that if the common compilers ever get a common (0,0) overlay, RNC could reside there, thus doing the job for everyone in exactly the same way.

The second attribute of a standardized line is that it has a full zero word EOL* mark. COMCROC returns a source line in C format**. Token generation and line handling in general can be considerably simplified if they don't have to deal with partial words. RNC will space (blank) fill the final word of a source line if the EOL mark is not on a word boundary. See figure 3.8. There are no known adverse side effects resulting from altering the actual source input by blank filling.

---

* End of Line.

** i.e. a source line can have a 12 thru 66 bit EOL mark.



Where a ":" represents 6 EOL bits.

Fig. 3.8. RNC end-of-line formats

Finally, every full zero word blank line* will be converted to a full word of blanks followed by a full zero word EOL mark. Performing this simple task takes all the grief out of handling these strange blank lines.

---

* Now a CDC standard. See DAP S1040.

3.3.2.2 CLN and Line Classification

All pre-entokening line processing is performed under the control of a single executive. CLN (Classify Line) is the executive and coordinates the following functions:

1. Statement line number extraction*.

2. Statement label extraction.

3. Line classification.

Line classification involves determining that a source line is one of 5 types:

1. an initial line of a statement.

2. a continuation line.

3. a comment line.

4. a blank (null) line**.

5. a C$ line.

Normal FORTRAN source lines (i.e. not SEQ) were originally designed so that line type could be determined by looking at columns 1 and 6.

---

* SEQ mode input only.

** ANSI states that blank or null lines are to be treated as comment lines for '76.

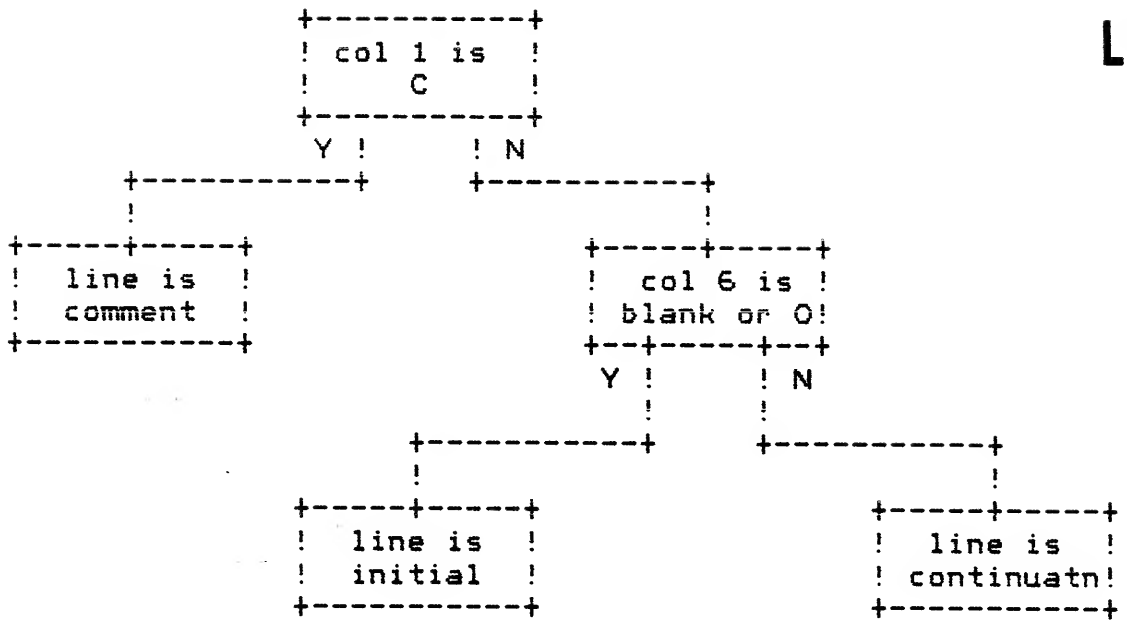


Fig. 3.9. Line typing hierarchy -original FORTRAN

Time has complicated this simple structure, as is shown in figure 3.10.

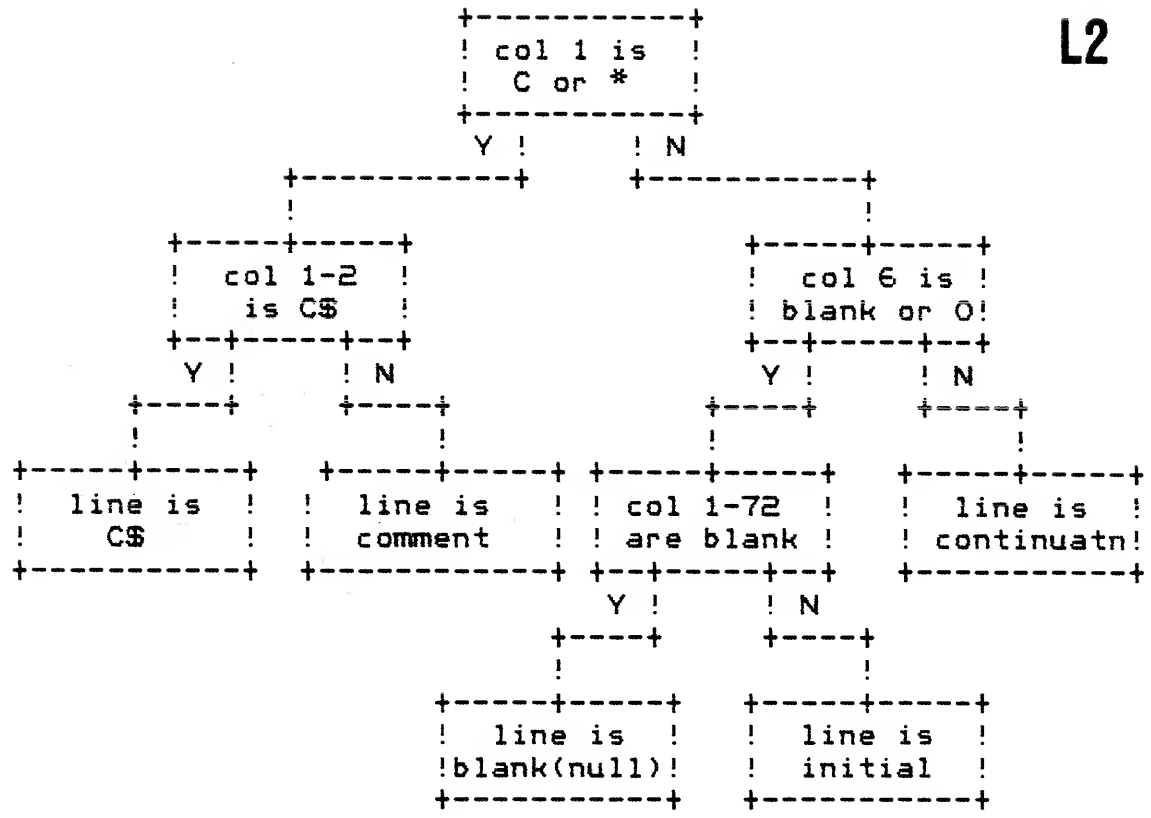


Fig. 3.10. Line typing hierarchy - FTN 5 (non-SEQ)

The most important difference between figures 3.9 and 3.10 is that one can no longer merely look at columns 1 and 6. EVERY column has to be scanned in order to type a line as blank. Consider also that, initially, a blank line "looks" like the initial line of a statement*. Now remember that line classification occurs at the tail end of LEX's main loop**, at a time when the statement in T.TB has not yet been compiled. This poses a problem in that we have to scan an entire line in order to type a blank line, but we don't want to do too much processing because looking at individual columns is in the realm of token generation. And in the case of an initial line, we don't want to generate tokens until the next cycle through the compiler master loop, after the current statement at T.TB has been compiled.

---

* i.e. column 6 is blank. Or in SEQ mode, both have a blank following the statement line number.

** See figures 3.5 and 3.6.

CLN solves this problem by using the BUB/BUN character access method* to strip the blanks that precede the statement keyword.

For example

```
          100         FORMAT(' HELP')

    col  1    6       15
```

The blanks in columns 7 thru 14 will be stripped, so that the token generator will begin at column 15. This means that, for a blank line, BUB will strip blanks until end of line. CLN can then sense this and properly type the line as blank.

SEQ mode "free form" input is a little different. Line number extraction, statement label extraction, line typing, and preceding blank strip must be performed left-to-right in the correct order.

Consider the line:

```
        100 200  FORMAT(' YOU ARE GETTING SLEEPY...')

    col 1  4    10
```

The line number "100" is extracted first, then the statement label "200", and finally the blanks in columns 8 and 9 are stripped. The line is typed as an initial, and when FEC reenters LEX to entoken this line, the entokener begins at column 10. In this sense, normal FORTRAN lines and SEQ mode lines are indistinguishable to the token generator**.

It was stated in the OVERVIEW section that the line number and statement label are stored in the locations TB=NUML/TB=NUMR and TB=LABL/TB=LABR. This is not done by CLN, however, because it is a low level executive which knows nothing about inter-line conditions. CLN stores this information into the cells:

LN=TYPE	line type
LN=NUM	line number
LN=LAB	statement label

---

* See section (3.3.3.1), Bub/Bun Character Access Method.

** This is not entirely true because normal FORTRAN lines are 72 columns wide, while SEQ mode lines are 80 columns wide.

The 5 possible line types (i.e. possible values of LN=TYPE) are defined by symbols that have the prefix "LT.":

value	symbol	meaning
1	LT.INIT	initial line
2	LT.CONT	continuation line
3	LT.CMNT	comment line
4	LT.NULL	blank or null line
5	LT.C$	C$ line

The scanner's main executive, LEX, transfers LN=NUM to TB=NUM and LN=LAB to TB=LAB when and if appropriate, and acts on LN=TYPE, thus maintaining a clean division of labor between detection (CLN) and control (LEX).

3.2.2.3 PLR and Listing

Very little has so far been said about listing. This is not to say that it is trivial, for many assume that listing is one of the scanner's simplest tasks. It is not. Few other places in the compiler can match its flag chasing abilities or pathologies. Beware . . .

This section is divided into 7 sub-sections:

Global Compiler Listing Logic

Begin at the top. This sub-section describes listing at the compiler level, primarily in terms of the listing flags. Understanding the listing flags and how they interrelate is half the battle.

PLINE and WOF

From the top to the bottom. This sub-section describes the only common ground in listing besides the listing flags: i.e. the low level routines that actually write a line.

Source Listing - Introduction

The deceptive middle ground. This sub-section describes some of the problems and structures involved in generating a source listing.

PLR - Process Listing Request

Describes LEX's executive for source listing.

NO LIST Mode

Describes one of the pathologies associated with listing . . . what to list when we're not supposed to be listing.

Before Header Mode

Another listing pathology . . . how to get the program unit name into the title line.

Deferred Listing File - An Opinion

Why not clean up this mess once and for all?

a. GLOBAL COMPILER LISTING LOGIC

Listing logic falls into 2 broad categories: what to do when a list option is selected, and what to do when a list option is deselected. The various list options are selected/deselected via the FTN control card and/or the C$ LIST directives. These options "translate" into a number of global compiler flags. At least this way, when confronted with the intricacies of listing, one can say, "Well that's because such and such flag is set to so and so." Onward . . .

There are 2 copies of the listing flags: a master and a working. The master copy represents the list options as selected on the FTN control card, and is set up by the control card cracker*. The working copy represents the current dynamic list options, as possibly altered by C$ LIST directives.

---

* See COMCPAC (Process Arguments from Control Statement) in deck INITOO (2.9.1).

The working copy is necessary because C$ LIST directives are only supposed to affect the program unit in which they occur. The master, which reflects the FTN control card, is copied to the working at the beginning of each program unit, thus destroying the effect of any C$ LIST directives that might have occurred in the preceding program unit. The master copy resides in deck FTN in the (0,0) overlay**, and the working copy resides in the deck PUC***.

Following is a brief discussion of the global list flags and how they determine/describe the type of listing to generate. With the exception of CP.LSTF, each flag is a master/working pair.

CP.LSTF

This is the master list flag (set via L control card option). CP.LSTF has to be ON for most of the list options to be able to take effect. The exceptions are the error list options. If an error of the requested level is detected in a source statement, the statement in error is listed along with its appropriate error message, regardless of the value of CP.LSTF. If L=0 is selected, the control card cracker will set CP.LSTF to OFF. Then when it has finished cracking the control card, it will set all the other list option flags (with the exception of the error list flags) to OFF.

WO.LOS

This is the source list flag. CO.LOS represents the LO=S control card option. The working WO.LOS is dynamic and can be affected by C$ LIST directives. If WO.LOS is OFF, then the compiler is said to be in NO LIST mode■.

** FWA is CO.C$.

*** FWA is WO.C$.

■ See 3.3.2.3 PLR and Listing/NO LIST Mode.

** note ** L7

If CO.LOS is OFF (i.e. L=0 or LO=-S selected on the control card), then C$ LIST directives will be prevented from affecting WO.LOS. This is because C$ LIST directives are never allowed to override options specifically deselected via the control card. See 3.3.2 Supporting details/C$ Statement Processing/LIST directives.

***

WO.LOS is used primarily to structure the "logical" relationship between the lexical scanner (which lists source lines in LIST mode) and the compiler's error processor (which possibly lists source lines in NO LIST mode). The coordination between LIST and NO LIST modes is of major concern in listing logic, and is discussed further throughout the remainder of 3.2.3 PLR and Listing.

b. PLINE_AND_WOE

PLINE (Print Line) is the macro* that is used to write a line to the OUTPUT file. This includes lines written for the source listing, error messages, reference map, object listing, and TEST mode compiler debugging output. Each PLINE reference expands into a call to WOF (Write Output File). WOF is the only real "top-down" executive associated with listing . . . a call to it will write a single "interesting" line to the OUTPUT file with an optional number of preceding blank lines.

Its main functions are to keep track of pages by outputting the source listing title lines at the appropriate times, to output preceding blank lines, and to invoke the I/O routines.

WOF actually writes a source line via the WRITEH macro**. WRITEH calls the MACE I/O comdecks on operating systems that support CIO, and calls the FA= comdecks on operating systems that do not support CIO. The FA= comdecks simulate CIO in a RECORD MANAGER environment. In a CIO environment, WRITEH calls COMCWTH which automatically strips trailing blanks from each line before writing it to the OUTPUT

---

* Defined in FTN5TXT.

** Defined in CPUTEXT for CIO (non-RECORD MANGLER), and in FTN5TXT for RECORD MANAGER.

file. This can considerably speed up output at a slow speed interactive terminal because then it does not have to print any trailing blanks that occur in a line. Interactive users rejoice . . .

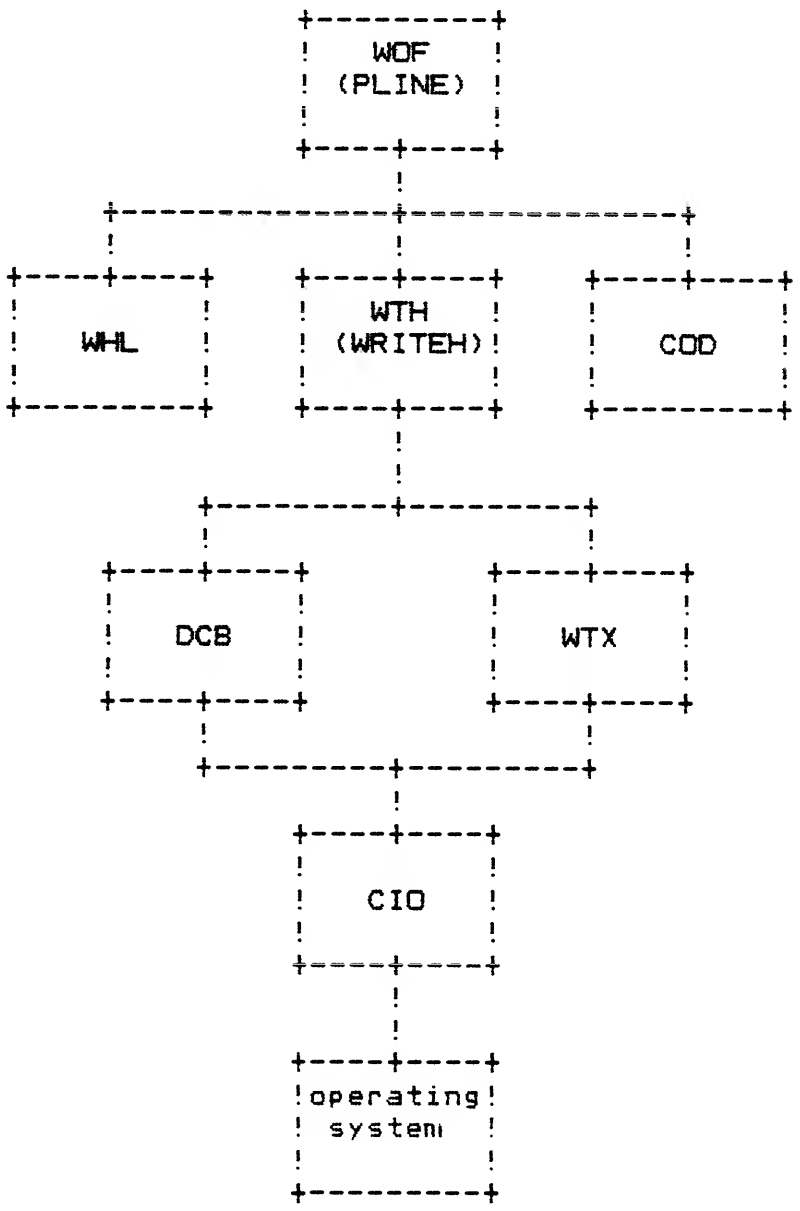


Fig. 3.11. Write routines hierarchy - (CIO)

The reader might find it interesting to compare figure 3.11 with figure 3.7.

PLINE and WOF, the lowest level executive in listing, are placed here early in the discussion of listing because they are really the only common element in listing . . . everyone calls WOF to write a line to the OUTPUT file. It is hoped that PLINE and WOF will therefore provide some "home ground" from which to embark on the following journeys.

c. SOURCE LISTING - INTRODUCTION.

Generating a source listing would seem to be one of FTN's simplest tasks. In fact, it is one of the most complex. The primary reason for this is that so many different conditions affect the way a source listing will look in its final form.

For example, did the programmer deselect a source listing (L=0, LO=-S, or C$ LIST NONE active)? What this really means is that a source listing is not generated only if there are no errors in the FORTRAN program, and this, in turn, depends upon the error detection level selected on the FTN control card (EL= option).

C$ LIST directives complicate the issue further. If a source listing was selected via a control card option, C$ LIST lines are always listed so that the programmer can know where and when her source listing was turned on and off*, EXCEPT when a C$ LIST NONE line is the 1st line of a program unit. In this case, nothing is listed (unless, of course, there's an error).

And there there's "before header" mode. Programmers like to have the name of the program unit in the title line of the listing for each program unit. This means that, in LIST mode, the compiler has to delay listing everything until the header statement processor (PROGRAM, SUBROUTINE, FUNCTION, or BLOCKDATA) has processed the header statement and extracted the program unit name from the token string (T.TB) and placed it into the title line.

Unfortunately, there is no one "master" control center in the compiler that can coordinate/structure all these tasks. Individual executives each control their own separate (or sometimes not so separate) part of the source listing logic. It might be useful to refer to figure 3.12 while reading the following sections.

---

* See 3.3.3.2, C$ Statement Processing/LIST directives.

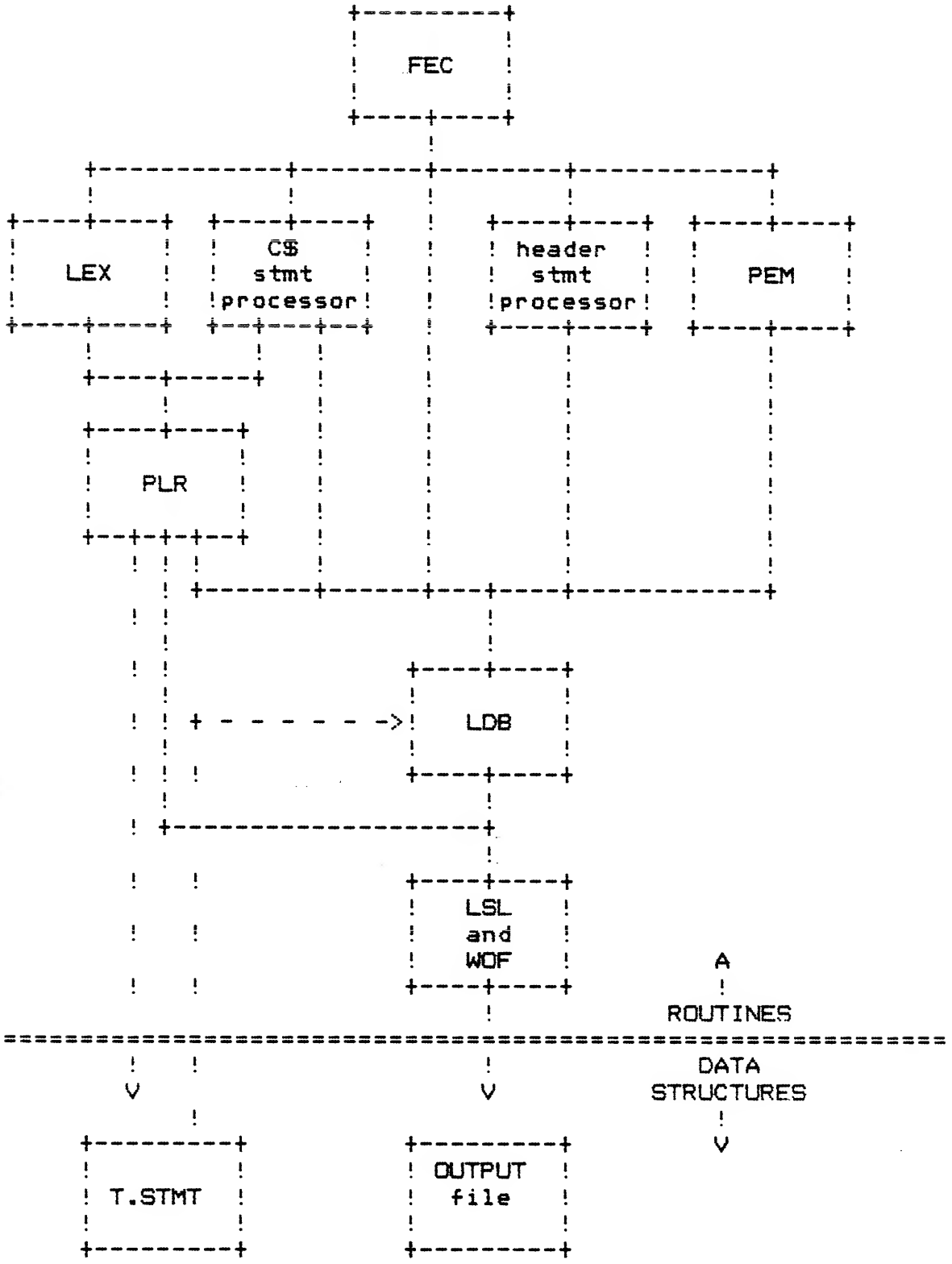


Fig. 3.12. Source listing routines structure

d. PLR - PROCESS LISTING REQUEST

PLR is the executive that is called whenever LEX is ready to "list" a source line. The word "list" is emphasized because the name PLR is somewhat of a misnomer. PLR is really not so much concerned with listing as it is with freeing up the space occupied by a source line at CP.CARD.

CP.CARD is a dual purpose data structure: source lines are read into it one-at-a-time and listed from it one-at-a-time. The problem is that this simple READ-LIST process is not always possible. For the "average" case, PLR does control listing, but in NO LIST and "before header" modes, PLR doesn't do any listing . . . it merely saves/accumulates source lines in T.STMT. Someone else makes the decision whether or not to list the lines at T.STMT.

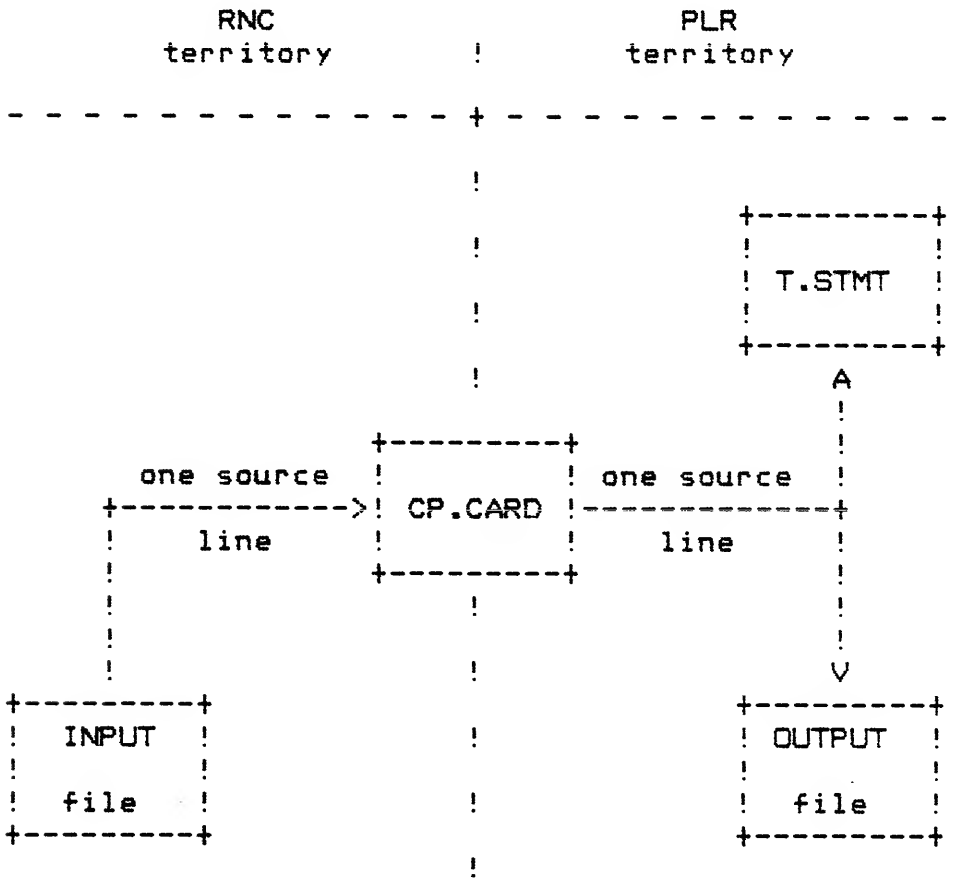
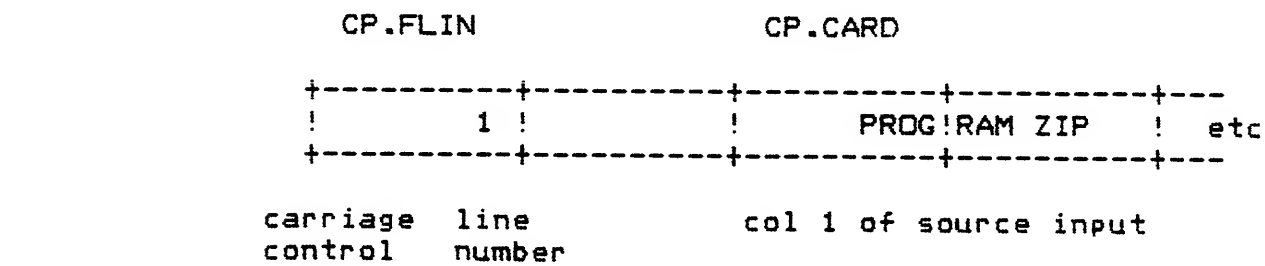


Fig. 3.13. CP.CARD and source line flow

Before PLR moves a source line from CP.CARD, it performs some line formatting. The two words preceding CP.CARD, beginning with CP.FLIN, are used for carriage control, compiler generated line number, and spacing.



A line number is generated only in non-SEQ mode (SEQ lines already have line numbers), and only in the following cases:

in LIST or "before header" modes

1. if the line number is a multiple of 5 (1,5,10, . . .etc), or . . .

2. if the line is a C$ LIST line, or . . .

3. if the contents of CO.SNAP are non-zero in TEST mode (for compiler debugging).

in NO LIST mode

1. if the line is an initial line of a statement, or . . .

2. if the line number of a continuation line is a multiple of 5.

C$ LIST lines are unconditionally given line numbers in non-SEQ mode so that the FTN programmer can see which lines are missing between a C$ LIST NONE line and a subsequent C$ LIST ALL line.

e. NOLISI

In NO LIST mode, FTN always lists source statements that are found to be in error along with their appropriate error messages, regardless of the source list options selected by the programmer. When a source statement is found to be in error, PEM (Print Error Message) in deck PEM is invoked to issue the appropriate error message.

In NO LIST mode, PEM is also responsible for listing the source line in error (which PLR saved in T.STMT). PEM does this by unconditionally calling LDB (List Deferred Buffer). In addition to listing the contents of T.STMT, LDB also takes care of some of the bookkeeping associated with T.STMT. For example, it prevents the possibility of T.STMT inadvertently getting listed twice by querying a status bit in the header word of the 1st line in T.STMT. This bit (defined by symbols SB.LISP and SB.LISL) is set by PLR when it puts lines into T.STMT only if they are also listed.

f. BEFORE_HEADER_MODE

In "before header" mode, PLR tries to save all source lines (including comment lines) in T.STMT, in an attempt to allow the header statement processor (PROGRAM, SUBROUTINE, FUNCTION, or BLOCKDATA) to place the program unit name into the title line of the source listing.

PLR will not save lines forever, though, because for programs that have large numbers of comment lines immediately preceding or following the header statement, T.STMT would soon usurp all of the managed table space. In the extreme case, the compiler would start "MEMing"* for more managed table space and soon usurp the entire machine. PLR will, therefore, only save lines while the number of lines in T.STMT is less than an arbitrary amount (defined by symbol MAX.LINC).

** note **

Using a "line" thresh-hold for T.STMT is different than the implementation in either FTN 4 compiler. They both use a word thresh-hold in determining when to terminate "before header" mode. A number of PSR's have been submitted against FTN 4 by users who did not understand why some of their programs got the program unit name into the title line and some did not. This is largely due to the fact that the number of lines that can occur within a given number of words in T.STMT is incredibly variable. It is hoped that by using a line thresh-hold in FTN 5, it will be easier for a user to understand the conditions under which a program unit absolutely will get the program unit name into the title line of the source listing.

***

* FTN "lingo" for making an operating system request for more central memory (via MEM RA+1 request).

"Before header" mode is automatically terminated whenever LDB is called. One of LDB's main functions is to keep NO LIST and "before header" modes from stepping on each other's toes. LDB is called when one of 5 things happens:

1. T.STMT becomes longer than MAX.LINC, or . . .

2. The header statement processor finds the program unit name in T.TB and places it into the source listing title line, or . . .

3. FEC (Front-end Controller) senses that this program unit doesn't have a header statement, or . . .

4. A C$ LIST(NONE) line is sensed (i.e. when entering NO LIST mode)*, or . . .

5. An error is detected and control ends up at PEM in deck PEM.

It is perhaps useful to note that "before header" and NO LIST modes are mutually exclusive. That is, you can be in one or the other, but not both.

g. DEFERRED LISTING FILE - AN OPINION

A deferred listing file is a data structure that basically has the format of a T.STMT for an entire program unit (or perhaps compilation). Each line in it would have a header word that contained information about the line. For example, the line type (initial, continuation, etc), whether the line was to be listed, etc. If a source listing of any kind was selected via the FTN control card, a deferred listing file would be generated. Then at the end of a compilation, it would be read back in, interpreted, and a source listing (to the OUTPUT file) generated from it.

---

* This is necessary to prevent overall confusion in pathological listing circumstances. See 3.3.3.2, C$ Statement Processing, CDDIR, (3.6).

Use of a deferred listing file would give the compiler total control over the bells and whistles that FTN programmers like to have in their source listing. Its primary disadvantage is that it slows down a compilation due to the added operating system (I/O) requests. For this reason, the powers that be opted not to use a deferred listing file in FTN 5, even though it would considerably simplify listing logic. This, I feel, in the long run will prove to be a mistake. I do not believe that speed of compilation (especially in LIST mode) is as important as the simplification of a major internal and external structure.

3.3.2.4 TOK and Token Generation

This section is divided into 4 sub-sections:

Overview

Provides a brief overview of token generation.

FTN Tokens

Describes the tokens that can be generated for a FTN program. It is hoped that by describing what TOK is trying to do (i.e. its output), that the rest of this section will come much easier.

Learning TOGEL

This sub-section is basically a tutorial in the language of token generation, TOGEL. What it does and how to use it.

Describing FORTRAN with TOGEL

Describes/explains the TOGEL program that describes FORTRAN.

TOK - Token Generator

Describes the inner workings of TOK, and how it actually generates tokens.

a. OVERVIEW

TOK is the executive that performs token generation. It is called once per cycle through LEX's main loop, and entokens a single source line. This source line resides in T.STMT, and is therefore in a packed (10 character per word) format with a full zero word EOL mark*. TOK generates tokens, one per central memory word, to the token buffer, T.TB.

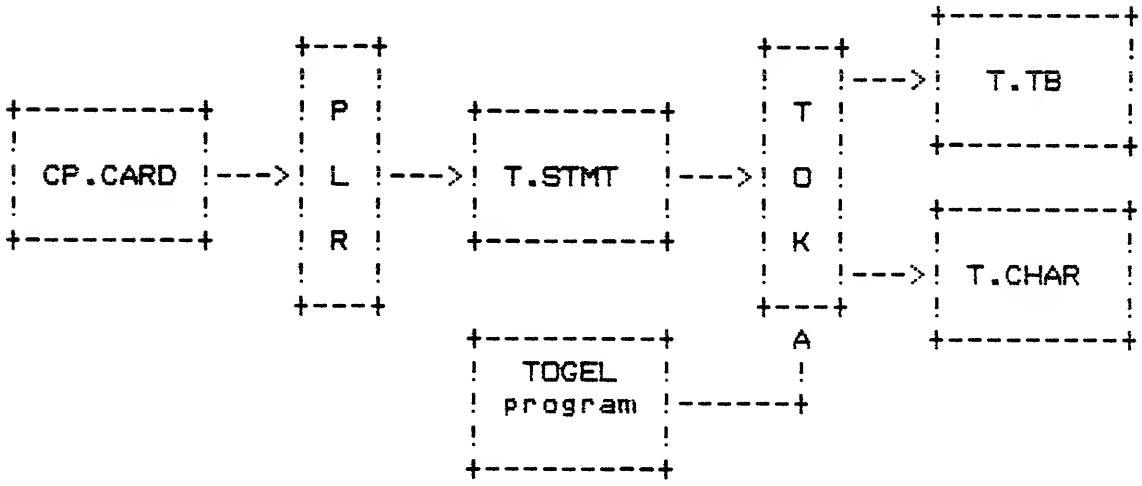


Fig. 3.14. Relationship of PLR to TOK

* The LEX executive, PLR, will have previously transferred the source line from CP.CARD to T.STMT. See 3.3.2.3 Design/Executives/PLR and Listing/PLR - Process Listing Request.

It is important to remember while looking at Figure 3.14 that T.STMT is not a complete, static data structure between PLR and TOK. For each cycle through LEX's main loop, PLR adds a new line to T.STMT. TOK is therefore always entokening the last line put into T.STMT.

TOK is designed to be a general purpose token generator. That is, it could just as easily be used to generate tokens for PL/I or COBOL as for FTN (this assumes, of course, that these compilers want to generate tokens for their source input). This is possible because TOK is table driven. Along with the actual source line to be entokened, it accepts as its input a table that describes how it is to generate tokens. This logic table is generated via a set of COMPASS macros that "look" like a simple "top-down" programming language. Therefore, these macros are called a "language", TOGEL (Token Generation Language), and when these macros are applied to a particular token generation problem, the result is called a "TOGEL program".

b. EIN_IOKENS

This sub-section is concerned with the general "form" which tokens can take, and how they structurally interrelate.

FTN5 tokens are a way of representing, or notating, FORTRAN source statements. A way that is easier for the statement processors to manipulate, and too tedious for a human to use. The middle ground.

The tokens for FTN 5 have been contrived so that they describe FORTRAN source statements in the most fundamental, statement-independent way possible. That is to say, so that tokens can be generated with no knowledge of what type of statement (PRINT, GOTO, replacement, etc) is being entokened.

This "mindless entokening" principle means that entokening is the process of grouping characters (into tokens) that have some simple structural, or syntactic, relationship to one another. For example, grouping alphanumeric characters into variable names, grouping numeric characters into numeric constants, grouping character data, and grouping the remaining

characters into operators. The structure of FTN's individual tokens is very simple. Each token occupies a single CM word and has the following general format:

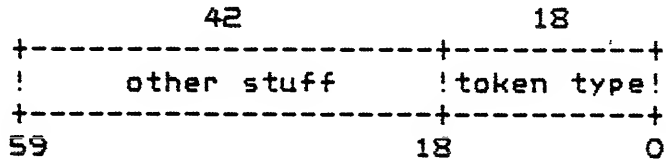


Fig. 3.15. FTN token format

Where "token type" is an 18 bit binary value used to distinguish one token from another. This number is defined via a symbol that has the prefix "O.", which stands for "operator/operand" (for years, I wondered what "O." stood for . . .). For example, the BOS (Beginning of Statement) token is defined by the symbol O.BOS (value = 0)*.

** note **

The lexical scanner can only generate 42 different tokens. This is <u>not</u> a scanner restriction, but a restriction imposed by the CONO table which is used by the parser, PAR in deck PAR. The CONO table is used to check the legality of a particular token following another. For example, to detect that in FORTRAN, an = token cannot follow a / token. The CONO table has only 42 bit positions for token types, ergo only 0 thru 41 token types are possible.

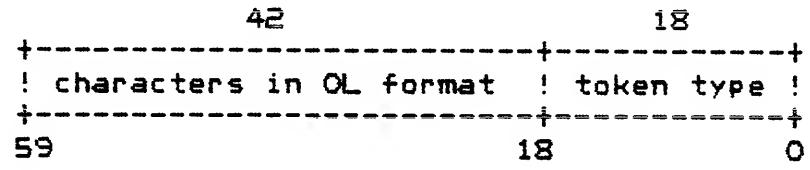
This does not imply, however, that <u>FTN</u> has only 42 different tokens. The parser, itself, can generate special tokens when it encounters certain syntax. For example, PAR will replace the O.LP token that precedes a FUNCTION argument list with a O.SLP token. This is done to simplify the parsing stack logic. See CONO table in deck PAR (B-3.12).

***

* "O." symbols are defined in FTN5TXT.

The primary virtue of this token structure is its simplicity, which happens to make TOK's job relatively straightforward. FTN5 tokens take on 3 generally different forms. These 3 forms differ both in physical layout and in the method in which they are generated.

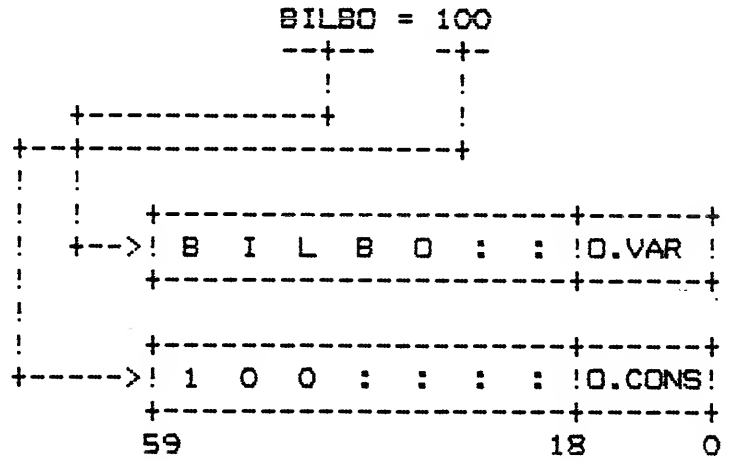
FORM 1 TOKENS (VAR, CONS)



Currently, the only 2 tokens that are of FORM 1 are the variable name token (O.VAR) and the numeric constant token (O.CONS). In this form, the token type in bits 0 thru 17 is really only an attribute of the characters in bits 59 thru 18. To better understand this, first consider which characters can become each of these tokens. An O.CONS token can consist of any of the characters 0 thru 9. An O.VAR can consist of any of the characters A thru Z and/or 0 thru 9, but the 1st character must be in the range A thru Z.

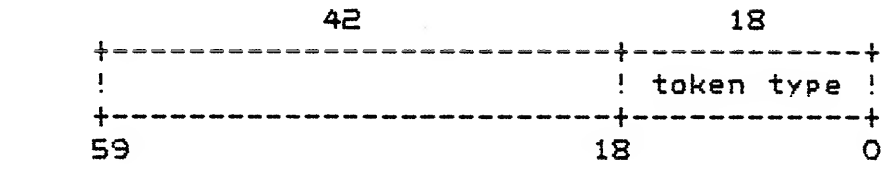
In the world of left-to-right scanning then, the primary difference between these two token types is whether the first character is numeric or alphabetic. This is an attribute of the characters that constitute the token, which in FORTRAN, happens to distinguish a variable name from a numeric constant.

FORM 1 Examples:



where : is zero fill (OOB).

FORM_2_TOKENS_(OPERATORS)



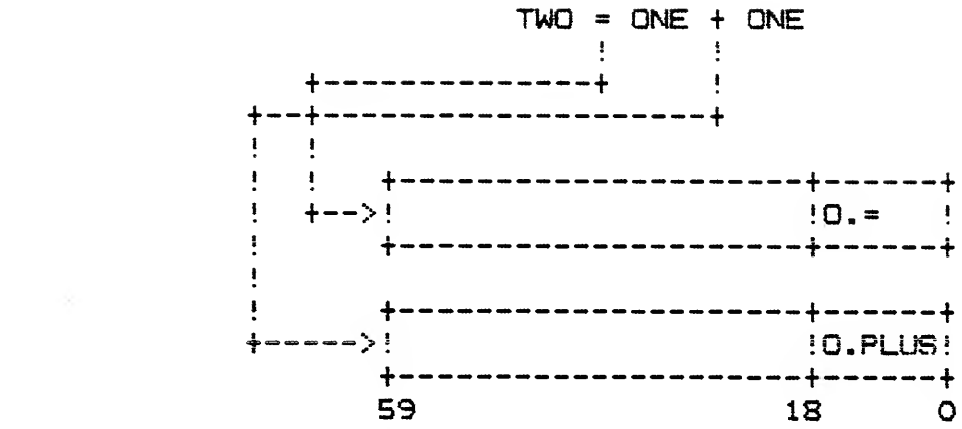
FORM 2 tokens are the FORTRAN operators (= - * etc). While in FORM 1 tokens, the original source characters still have "value", in FORM 2 tokens, they do not. The token type itself represents the sum total of the information the scanner is passing on to the statement processors. Stated more directly, each FORTRAN operator has its own token type (O. symbol).

It is perhaps interesting to note that this need not have been so. FORTRAN operators could have been represented as FORM 1 tokens, where the token type was "operator", and the actual characters that constitute the operator occur in bits 59 thru 18. This would, however, significantly complicate the work of the statement processors, and is not practical in light of FTN's overall design.

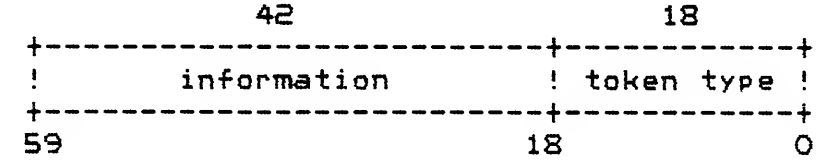
LEX does not pass anything of value to the statement processors in bits 59 thru 18 of FORM 2 tokens. These bits are used, however, by the token generator during generation of these tokens. Because the token generator is extremely table driven, these bits contain information set up at assembly-time by the TOGEL macros that describe "how" to generate these tokens.

If the reader is making a "first pass" across this document, the details of how these bits are used is not important at this point in time. For more information, see the following subsections entitled, Learning TOGEL, and TOK - Token Generator.

FORM 2 examples:

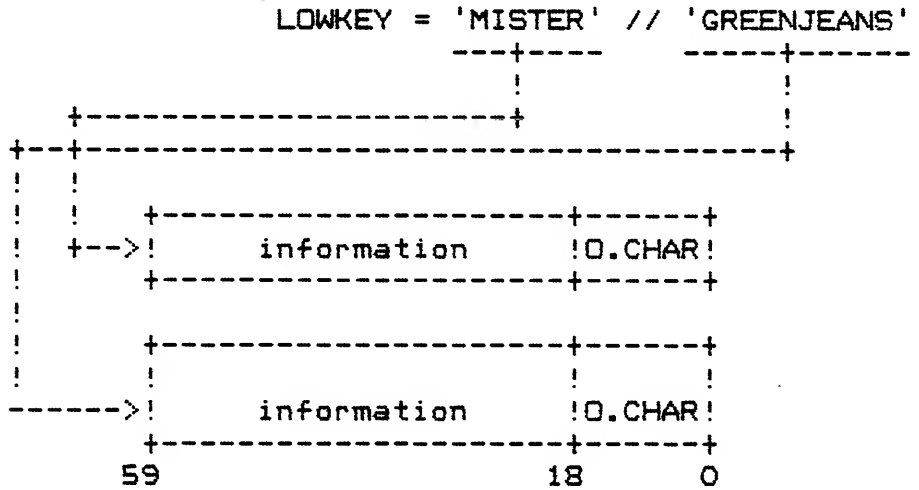


FORM 3 TOKENS (CHAR, HOLL, OCT, HEX)



FORM 3 tokens are the character data tokens, O.CHAR and O.HOLL. Bits 59 thru 18 contain information about the character data, i.e. the T.CHAR ordinal of the actual character string, the number of words and characters in the character string, etc*.

FORM 3 examples:



---

* See DESCRIBE/DEFINE definitions of "TB." symbols in FTNTEXT for more information about character token formats.

TOKEN_VS_TOKEN

Let us now consider a few of the more fundamental relationships that tokens have between each other in the token buffer (T.TB). Consider the statement:

```
WAYTOOLONG = ((A+1)*2)
```

The variable name WAYTOOLONG is way too long for a single O.VAR token, which can only contain up to 7 characters.

Consequently, WAYTOOLONG becomes 2 tokens:

```
+---------------------+------+
! W  A  Y  T  O  O  L !O.VAR !
+---------------------+------+

+---------------------+------+
! O  N  G  :  :  :  : !O.VAR !
+---------------------+------+
```

This means that the statement processors have to be smart enough to know that multiple O.VAR tokens represent a single variable name. This "overflowing" is only possible in FORM 1 tokens, i.e. with variable name (O.VAR) and numeric constant (O.CONS) tokens.

Figure 3.16 shows the entire statement entokened.

M7

WAYTOOLONG = ((A+1)*20)

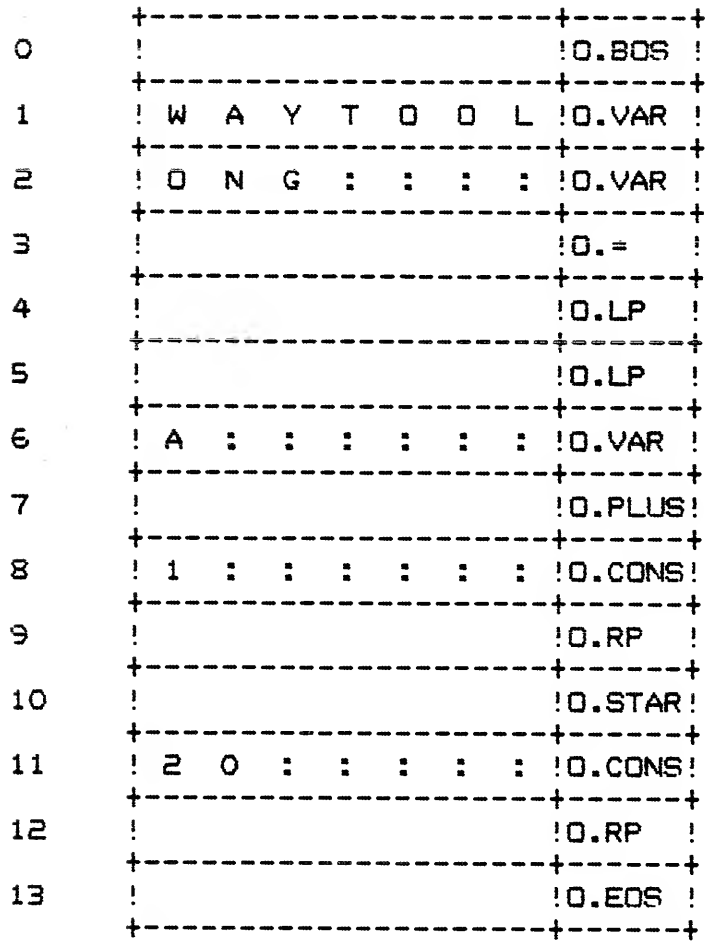


Fig. 3.16. Example of an entokened statement

Note via figure 3.16 that, in contrast to the double O.VAR tokens which represent one variable name, the double O.LP tokens (ordinals 4 and 5) will be interpreted by the statement processors as being 2 left parentheses.

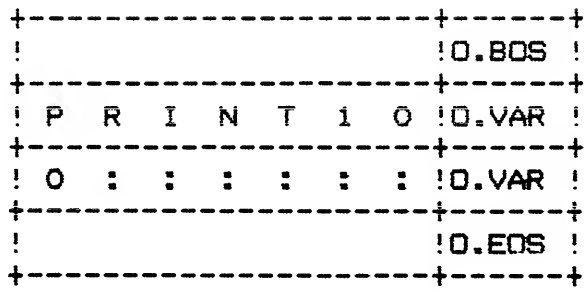
KEYWORDS

No distinction is made between FTN keywords and variable names during token generation. Keywords are variable names as far as TOK is concerned.

Consider:

```
    PRINT 100
```

which becomes



The distinction between keyword/variable name and between PRINT/100 is made later when the statement typing mechanisms are invoked*. This is done to free TOK from having to concern itself with FORTRAN's down-right weird keyword syntax.

Consider the confusion that could arise during token generation if all keywords were to be special cased:

```
PRINT 1,AHHH        keyword
      vs
PRINT1 = AHHH       replacement

IF(YECH) GOTO100    2 keywords
      vs
IF(YECH)=GOTO100    replacement

          and on, and on, . . . .
```

c. LEARNING TOGEL

TOGEL (Token Generation Language) is the set of COMPASS macros, or language, if you prefer, for describing token generation. Visually, it looks somewhat like any one of the glut of popular "top-down" programming languages on the market today.

---

* See 3.3.2.5 Design/Executives/CST and Statement Typing.

This sub-section is very tutorial in nature. It does not discuss how TOGEL works internally (i.e. how TOGEL is implemented). If looking for the "hows" of implementation, see 3.2.4 Design/Executives/TOK and Token Generation/TOK - Token Generator, and see 3.3.3 Design/Supporting Details/Using COMPASS to Compile TOGEL.

```
GROUP (0..9)
IFT (HLR)
   CALT XXX
   GROUP (-..)
ELST
 .
 .
etc
```

Fig. 3.17. Sample TOGEL program

The main difference between TOGEL and all those other languages is that you can't do anything with TOGEL but generate tokens. It is very specifically oriented to the generation of FTN tokens (handy, huh . . .).

Before diving into the mechanics of <u>how</u> to use TOGEL, one might be interested in the <u>why</u> to use TOGEL. Why complicate token generation by requiring the learning of one more eminently forgettable language? There are 2 reasons: flexibility (i.e. fixability), and conciseness. Token generation is difficult to read and understand in COMPASS because it is, by its very nature, <u>very</u> tight code. Consequently, the reader-of-code becomes inundated with the details of registers and low level logic. TOGEL allows one to take a small step (but a step nonetheless) back so that one can see all the logic of token generation in a single eyeful. In this way, it is easier for a human to critically analyze the problems of token generation, and to solve them at a sufficiently high enough level that they will remain solved (i.e. not just buried).

TOGEL does not pretend to be complex or esoteric. It is <u>very</u> simple. TOGEL "instructions" are "executed" sequentially in the fashion that all programmers are familiar with. In fact, each TOGEL instruction/macro <u>could</u> be replaced with actual COMPASS instructions (although they're not, primarily for space-efficiency reasons).

TOGEL has one verb: GROUP; a number of flow control mechanisms: IFT, ELST, ENDT, CALT, and GOTO; and the special entokening mechanism: CASEOF, TOKEN, and ENDC. The remainder of this sub-section discusses each of these "instructions" in detail.

GROUP

The GROUP verb can be thought of as a character function. The basic idea behind it is the ability to "group" consecutive characters in a left-to-right scan. The characters, or range of characters, that can occur within a particular "group" are specified as an argument in the GROUP verb.

Consider a line consisting of the characters

```
AABAAACBD
```

First, an invisible, internal pointer is set to the beginning-of-line (i.e. to the 1st character in the line).

```
AABAAACBD
!
```

The command

```
GROUP (A)
```

which reads, "group all the A characters", produces:

group #	contents	source line
1	AA	AABAACBD !

Now the command

```
GROUP (AB)
```

which reads, "group all the A and B characters", produces:

group #	contents	source line
1	AA	AABAAACBD !
2	BAAA	AABAAACBD !

Alternately, given:

M11

```
AABAAACBD
!
```

the command:

```
GROUP (A..C)
```

which reads, "group the characters A thru C", produces:

group #	contents	source line
1	AABAAACB	AABAAACBD
		!

The 2 characters .. are interpreted as indicating a range of characters.

For example

```
GROUP (A..CK..N)
```

is read, "group the characters A thru C, and K thru N (i.e. ABCKLMN)". Character ranges are circular, that is

```
GROUP ( ;:A)
```

is equal to

```
GROUP ( ..A)
```

In addition, the .. operator has a default range. If the "from" operand is missing, its default is the 1st character in the character set, ":", and if the "to" operand is missing, its default is the last character in the character set, ";".

Therefore:

```
GROUP (..Z)       equals      GROUP (:..Z)
GROUP (A..)       equals      GROUP (A..;)
GROUP (..)        equals      GROUP (:..;)
```

Besides the .. operator, TOGEL has another special operator. If a - character occurs as the first character in a GROUP specification, then the entire specification is logically negated.

For example

```
GROUP (-AB)
```

reads, "group everything but A and B", which is equivalent to

```
GROUP (C..:)
```

The GROUP command can also be invoked with or without blank squeeze. If the NSQZ option/parameter is specified, then blanks (55B) will not be ignored (default is SQZ, i.e. blanks ignored).

For example, given:

```
A B C D
```

the statement

```
GROUP (A..C),,SQZ
```

would produce:

```
      group #        contents        source line

         1             ABC           A B C D
                                           !
```

While the statement

```
GROUP (A..C),,NSQZ
```

would produce:

```
      group #        contents        source line

         1               A           A B C D
                                      !
```

Now consider the statement:

```
THE=COW=JUMPED=OVER=THE=MOON
```

Keeping the structure and format of FTN tokens in mind, it can be seen that it is desirable to "group" these characters in the following manner:

characters/groups	tokens	
THE	O.VAR	variable name
=	O.EQUAL	equal operator
COW	O.VAR	variable name
=	O.EQUAL	equal operator
JUMPED	O.VAR	variable name
=	O.EQUAL	equal operator
OVER	O.VAR	variable name
=	O.EQUAL	equal operator
THE	O.VAR	variable name
=	O.EQUAL	equal operator
MOON	O.VAR	variable name

This can be performed with the following TOGEL program:

	group #	contents
GROUP (A..Z)	1	THE
GROUP (=)	2	=
GROUP (A..Z)	3	COW
GROUP (=)	4	=
GROUP (A..Z)	5	JUMPED
GROUP (=)	6	=
GROUP (A..Z)	7	OVER
GROUP (=)	8	=
GROUP (A..Z)	9	THE
GROUP (=)	10	=
GROUP (A..Z)	11	MOON

which can be optimized as follows:

```
LOOP  GROUP (A..Z)
      GROUP (=)
      GOTO LOOP
```

It is pretty easy to see that, in FTN5, each GROUP group is destined to become a token. As each source line is scanned from left-to-right, characters are built into groups that become tokens. So, in general, a single type of token is generated for each "execution" of the verb GROUP.

** note **

A distinction is made here between "a single type of token" and "a single token". A single type of token may consist of more than one consecutive token (i.e. consist of more than one CM word), all of which will have the same token type (i.e. "0." value in bits 0 thru 17).

Also, the words "in general" are used because there are exceptions to the rule in FORTRAN, particularly when dealing with tricky syntactic tokens such as the H, L, or R character data representations.

***

The token type (0. symbol) that a GROUP group is to have can be specified as the 2nd parameter on the GROUP command/macro:

```
GROUP (0..9),CONS
```

If a token type is specified, the bits 0 thru 17 of the token to generate will contain the token type (0.CONS in the above example). Bits 59 thru 18 will contain the "grouped" characters*.

FLOW CONTROL = IFT, ELST, AND ENDT

TOGEL is intended to be used as a block structured language, where the IFT, ELST, and ENDT mechanisms work in the usual manner (the suffix T is used to avoid conflicts with the COMPASS pseudo-ops).

IFT is used to test the character that terminated the most recent GROUP group.

---

* See previous sub-section entitled FTN Tokens for a description of FTN's tokens.

For example, given:

```
BYE = YALL
!
```

then the command:

```
GROUP (A..Z)
```

produces:

group #	contents	source line
1	BYE	BYE = YALL !

The "=" character terminated the GROUP (A..Z). This character can now be tested for via the IFT command:

```
IFT (=)
```

or perhaps via the more general:

```
IFT (+..:)
```

where the character or range of characters to test for is specified. IFT character ranges have the same syntax as GROUP character ranges, and the following THEN-ELST-ENDT block ranges are conditionally "executed" in the appropriate manner.

Consider the 2 different source lines:

```
BYE = YALL          BYE123
!                   !
```

The following TOGEL program allows the programmer to do one of two things:

```
line 1  GROUP (A..Z)
     2  IFT (=)
     3      GROUP (-..)
     4      GROUP (A..Z)
     5  ELST
     6      GROUP (0..9)
     7  ENDT
```

For BYE = YALL:

group #	contents	source line
1	BYE	BYE = YALL
2	=	BYE = YALL
3	YALL	BYE = YALL

Or for BYE123:

group #	contents	source line
1	BYE	BYE123
2	123	BYE123

** note **

Concerning the character range in GROUP (-..), line 3 of the preceding TOGEL program: Because of the internal timing involved with "grouping" characters, the GROUP command always forms a group of at least one character. Therefore GROUP (-..), which reads, "group not everything (i.e. nothing)", forms a group that contains only one character. Another way to look at this is to consider that a character that terminates a GROUP group (testable via IFT) always becomes the 1st character of the next GROUP group.

This inconsistency exists for efficiency (i.e. speed) reasons. The GROUP processor does not want to have to test for the null group before "grouping". This can be considered similar to the zero trip DO loop problem in FORTRAN. See Design/Executives/TOK and Token Generation/TOK - Token Generator.

***

GOTO_AND_CALL

The first of these two instructions is pretty self-explanatory. GOTO will transfer control to the specified label within a TOGEL program.

For example:

```
LABEL  GROUP (A..C)
         .
         .
       GOTO LABEL
```

The CALT instruction is used to invoke special user code when situations arise where the programmer wants or has to use COMPASS to get the job done.

For example:

```
    CALT PUPPIES
```

will force the token generator to suspend "reading" TOGEL instructions and transfer control of the CPU to the address PUPPIES. When the programmer's own code is finished doing whatever, it should transfer control to TOK=MN. This will return control back to the token generator and the TOGEL program in progress.

For example:

```
    GROUP (0..9)
    IFT (HLR)
        CALT TOK=HLR
        GROUP (..),,NSQZ
    ELST
      .
     etc
```

This TOGEL could be used to process the H, L, or R data representations which use an explicit character count, as in: 13HHELLO...HELLO. The code at TOK=HLR will convert the previously entokened character count (13 in our example) to binary, and then "dink" up TOK's pointers so that the GROUP (..),,NSQZ will group the appropriate characters (HELLO...HELLO in our example).

Registers are <u>very</u> tight in TOK and it is the CALT user's responsibility to make sure critical registers don't unwittingly get clobbered. Be careful.

COMCTOK contains 2 utility subroutines that can help alleviate some of the "tight register" problem. They are: SER (Save Entokening Registers) and RER (Restore Entokening Registers). It is probably a good practice to use SER and RER for all CALT user owncodes. It is IMPORTANT to note, however, that SER and RER do NOT save and restore register A6 (next address to store a token). This is due to the fact that it is not always possible to tell whether the "next token to store" is at (A6) or (A6+1). See SER and RER in COMCTOK.

CASEOF, TOKEN, AND ENDC

These instructions are used together and essentially define a character map technique for generating certain tokens (usually simple operators) as quickly as possible.

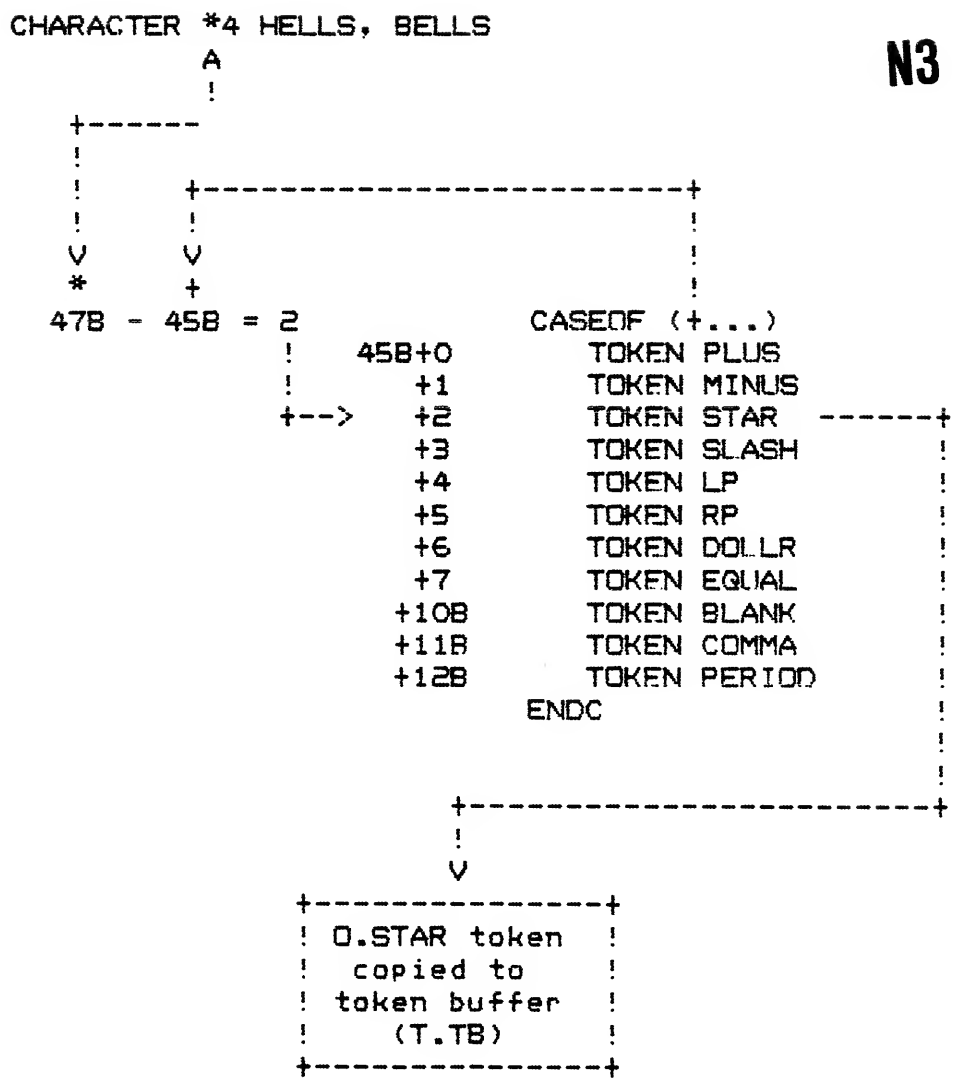
Their structure is as follows:

```
CASEOF (C1..Cn)
    TOKEN token type 1
    TOKEN token type 2
      .
      .
    TOKEN token type n-1
    TOKEN token type n
ENDC
```

where CASEOF and ENDC define the boundaries of the character map, and each TOKEN reference defines a token which can be potentially copied to the token buffer.

It works like this. . . First, the CASEOF statement contains a range of characters (C1..Cn) that specifies the first and last characters in the character map to follow. When the CASEOF statement is encountered in a TOGEL program, the binary value of the character that TOK is pointing to is biased by C1 and then used as an ordinal into the character map. That character map (TOKEN) entry contains a skeleton token which is picked up and copied to the token buffer. Voila . . . a token is born. The token type (O. symbol) that defines each token in the character map is specified as a parameter on each TOKEN instruction/macro.

Consider the following pictorial representation of how CASEOF-TOKEN-ENDC works for a particular example:



N3

Fig. 3.17a. Example of CASEOF-TOKEN-ENDC char map

Now consider:


```
    LOOP    GROUP (A..Z),VAR
            CASEOF (+...)
45              TOKEN PLUS      +
46              TOKEN MINUS     -
47              TOKEN STAR      *
50              TOKEN SLASH     /
51              TOKEN LP        (
52              TOKEN RP        )
53              TOKEN DOLLR     $
54              TOKEN EQUAL     =
55              TOKEN BLANK
56              TOKEN COMMA     ,
57              TOKEN PERIOD    .
            ENDC
            GOTO LOOP
```

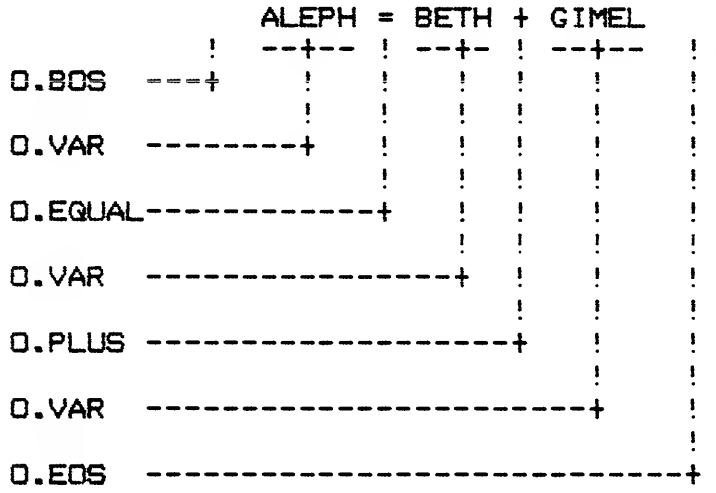
Fig. 3.18. TOGEL example

which could be used to entoken a hypothetical language that consisted of alternating variable names and simple, single character operators.

Consider a statement in such a language:

```
ALEPH = BETH + GIMEL
```

The TOGEL in figure 3.18 would produce the following tokens:



The O.BOS and O.EOS tokens are always invisibly generated and act as delimiters for the entokened statement.

The CASEOF, TOKEN, ENDC mechanism always advances TOK's internal pointer that points to the next character in the source line.

For example, given:

```
ALEPH = BETH + GIMEL
      !
```

then the statements:

```
CASEOF (+...)
    TOKEN PLUS
        .
        .
    TOKEN POINT
ENDC
```

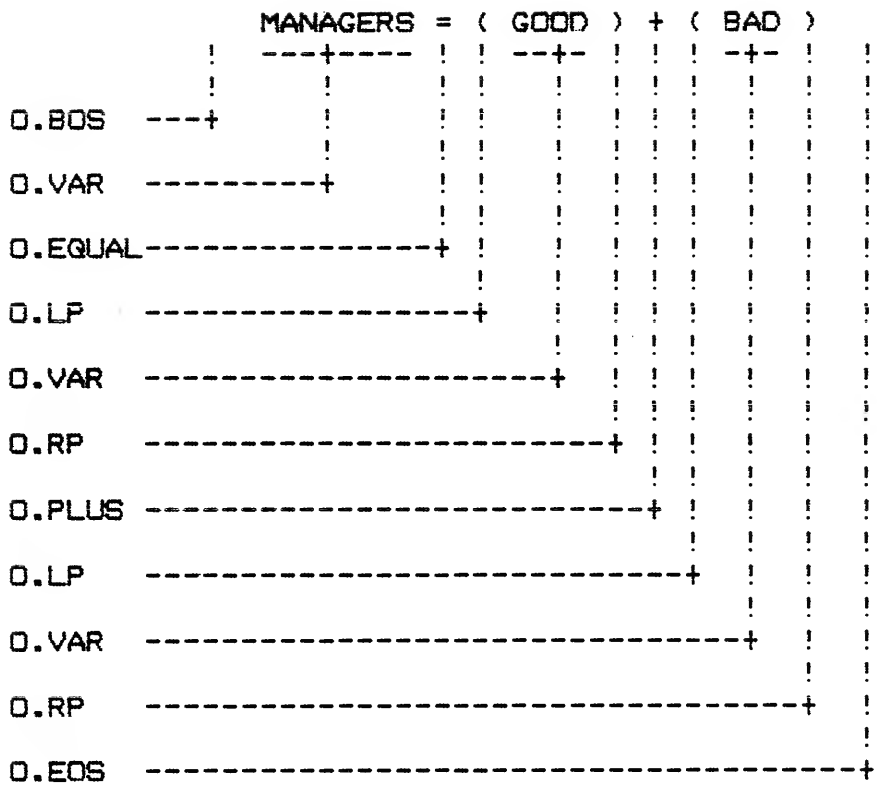
generate an O.EQUAL token, and result in:

```
ALEPH = BETH + GIMEL
        !
```

Our simple language of variable names and single character operators can now be expanded to:

```
LOOP    IFT (+...)
        THEN
            CASEOF (+...)
                TOKEN PLUS          +
                TOKEN MINUS         -
                TOKEN STAR          *
                TOKEN SLASH         /
                TOKEN LP            (
                TOKEN RP            )
                TOKEN DOLLR         $
                TOKEN EQUAL         =
                TOKEN BLANK
                TOKEN COMMA         ,
                TOKEN PERIOD        .
            ENDC
        ELST
            GROUP (A..Z),VAR
        ENDT
        GOTO LOOP
```

which entokens syntaxes involving consecutive single character operators, such as in:



Now suppose our hypothetical language is to have multiple character operators, such as ** exponentiation. This can be encoded in TOGEL via a 2nd parameter on the TOKEN instruction/macro. This 2nd parameter is used to describe multiple character operators in terms of other tokens. Emphasize tokens.

For example, to describe ** exponentiation, use:

```
CASEOF (+...)
   TOKEN PLUS                    +
   TOKEN MINUS                   -
   TOKEN STAR                    *
      TOKEN EXP,(STAR,STAR)      **
   TOKEN SLASH                   /
     .
     .
ENDC
```

This TOGEL states that the O.EXP token is defined to be 2 consecutive O.STAR tokens.

** note **

Understanding why multiple character operators are described in terms of tokens and not characters requires some knowledge of the internal workings/philosophy of TOK. For more information, see Background/Problems and Solutions/Multiple Character Operators, and see Design/Executives/TOK and Token Generation/TOK - Token Generator.

In any case, I do not believe that this is the appropriate place to delve into this subject. Please merely accept for the time being that describing multiple character operators in terms of other tokens is the most desirable (i.e. optimal) approach to this problem.

***

Some multiple character operators cannot be fully described in terms of other tokens. For example, consider the boolean operators .AND. and .OR., which both consist of the tokens O.PERIOD, O.VAR, O.PERIOD. The critical difference is the characters that constitute the O.VAR tokens. The TOKEN macro has a special syntax for these fellows:

```
CASEOF (+...)
   TOKEN PLUS
     .
     .
   TOKEN PERIOD
      TOKEN AND,(PERIOD,VAR'AND',PERIOD)
      TOKEN OR,(PERIOD,VAR'OR',PERIOD)
ENDC
```

where the actual characters to check for are enclosed in ('').

Specifying the actual character string check via the (") syntax is only meaningful when used with FORM 1 tokens*. If used with FORM 2 or FORM 3 tokens, unpredictable, or rather, unusual results can be expected. Don't do it.

---

* See the previous sub-section entitled FTN Tokens for a description of FORM 1, 2 and 3 tokens.

d. TOK - TOKEN GENERATOR

TOK is the executive for token generation, and exists as a common comdeck, COMCTOK. It accepts as its input a source line to entoken, and a table generated at assembly time by the TOGEL macros* that describes how to entoken. If one thinks of TOGEL as a pseudo compiler (written in COMPASS's macro language), then this table can be thought of as TOGEL's "object module", called TOM for short.

Each TOM consists of a sequence of binary TOGEL instructions that can be "executed" by TOK. Each binary TOGEL instruction occupies a single CM word and has the following general format:

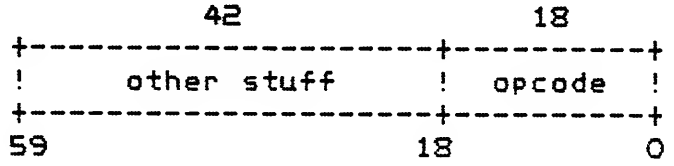


Fig. 3.19. TOGEL binary instruction format

A typical TOM, therefore, would take the form:

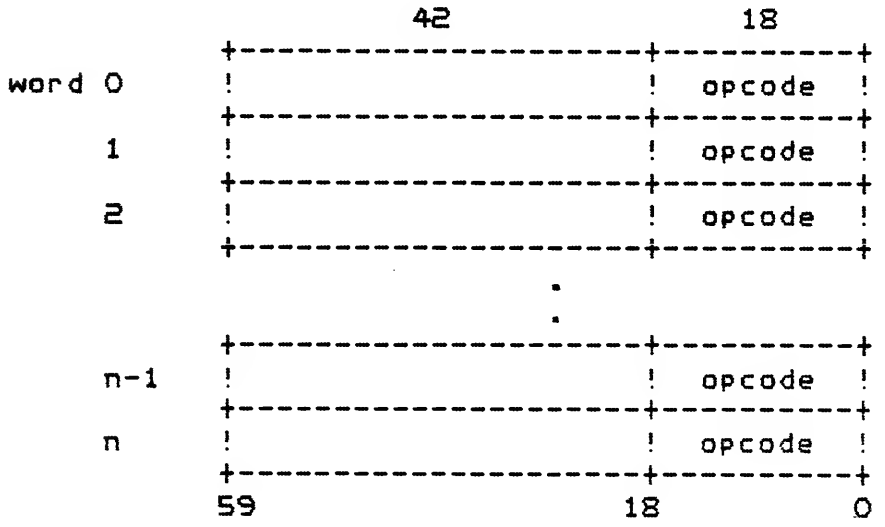


Fig. 3.20. TOGEL object module (TOM) format

---

* TOGEL macros exist in comdeck COMATOK. See previous sub-sections in this section, 3.2.3 Design/Executives/TOK and TOKEN Generation, for more information about TOGEL.

TOK will step through the TOM it was given, "executing" each binary TOGEL instruction as it is encountered.

TOK can therefore be thought of as a machine that executes binary TOGEL instructions from a TOM conceptually much in the same fashion that a hardware CPU executes a program object module. In "da bizniz", a software program that performs in this way is generally called an interpreter.

TOK is structured with a main driver, TOK=MN, surrounded by functional units (TOFUs) which do most of the work. TOK=MN acts as a focal point during token generation, and is responsible for reading instructions from TOM and farming each one out to its appropriate TOFU. TOFUs are totally independent of one another: there is one TOFU for each binary TOGEL instruction that the TOGEL macros can generate.

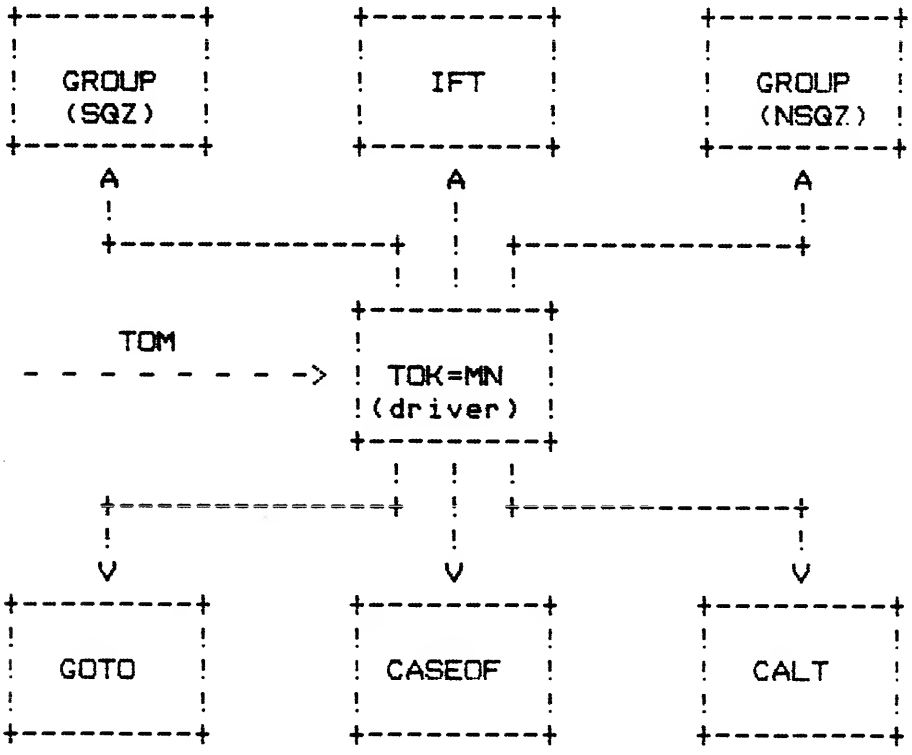


Fig. 3.21. TOK's structure, showing TOFUs

After a TOFU has performed its specific task, it will return control back to TOK=MN so that the next "instruction" can be "read" from TOM. This process of "reading" and "interpreting" continues until the entire source line has been entokened, at which time TOK returns control back to the caller.

** note **

Only 6 binary TOGEL instructions . . . what happened to ELST-ENDT, and TOKEN and ENDC? The ELST-ENDT structure gets converted by the TOGEL macros into GOTOs in the normal way (i.e. via the standard compiler cop out).

For example:

```
    IFT (A..Z)
        GROUP (A..Z)
    ELST
        GROUP (0..9)
    ENDT
```

becomes:

```
                          T     F
        IFT (A..Z)      --+   --+
                          !     !
        GROUP (A..Z)    <-+     !
        GOTO .2                 !
                                !
   .1   GROUP (0..9)    <-------+

   .2     etc
```

The ENDC macro is merely used to force assembly of its preceding TOKEN macros (i.e. it is required because of the way the macros work). It generates no code. See 3.3.3 Design/Supporting Details/Using COMPASS to Compile TOGEL.

The TOKEN macro does generate "code" to the TOM, that is processed/used by the CASEOF TOFU. In fact, TOKEN can be thought of as a sub-structure of the CASEOF instruction. See the sub-sub-section in this sub-section entitled, TOK=COF - CASEOF Instruction.

***

In the interest of clarity, and at the risk of being redundant, I would like to emphasize that binary TOGEL instructions specify operations that are to be performed <u>by</u> TOK <u>upon</u> the input source line. In a crude sort of way, they represent the <u>rules</u> that are to be used to translate/convert any input source line from its packed (10 character per word) form to its entokened form.

TOK works very closely with the BUB/BUN character access method which is used to make the actual character accesses upon the input source line. In the "lingo" of TOGEL, these routines are capable of GROUPing characters in an entokenable form. TOK's various TOFUs will call BUB and/or BUN when they need characters from the input source line. BUB and BUN, therefore, are slaves to TOK's various TOFUs, which are in turn, slaves to the TOM driving them.

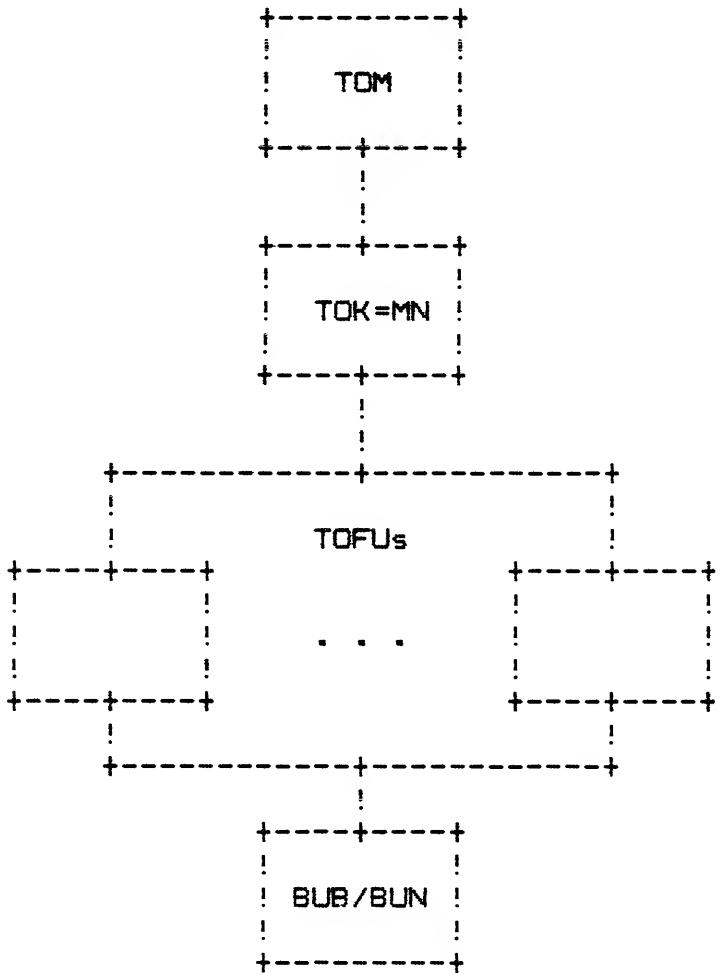


Fig. 3.22. Entokening routines hierarchy

INTERNAL POINTERS/REGISTERS

TOK has a set of global internal pointers that describe where it is entokening (i.e. where we are in the input source line), where it is entokening to (i.e. somewhere in T.TB), and where it is in the TOM that is running the show.

These pointers are kept in registers whenever possible, and are saved in memory when necessary. These pointers are the reason that TOK is so tight on registers. Registers are used as much and as long as possible so that TOK will run as fast as possible.

A COMCTOK caller provides a parameter list, called TOKCOM (TOK communications area), that contains the initial values of these internal pointers/registers. One of the first things TOK does is to transfer appropriate TOKCOM entries into TOK's global registers. During the entokening of a source line, TOKCOM is used as a place to dynamically materialize/save these global registers when necessary. Registers are materialized/saved in TOKCOM when either registers get too tight to get the job done, or if COMCTOK is running in TEST (i.e. debug) mode. In TEST mode, TOKCOM is updated in TOK's main loop, TOK=MN, before every TOGEL "instruction" is executed. This feature can provide an "audit trail" that can be useful in debugging.

Because TOK has a dissociated internal structure (i.e. TOFU's are invoked independently of one another, based entirely on what is in the TOM being executed/interpreted), these internal pointers constitute a common bond between all the parts.

Most of TOK's global pointers/registers are also BUB/BUN global registers (i.e. COMCTOK and COMCBUB/COMCBUN work hand-in-hand very closely, and share common registers).

The only register that is unique to TOK (i.e. BUB/BUN don't know anything about it, except that they must leave it alone) is A0. Register A0 is TOK's pseudo P register. It points to the next binary TOGEL instruction to execute in TOM. Therefore, when within a particular TOFU, the instruction being executed/interpreted is at A0-1.

N16